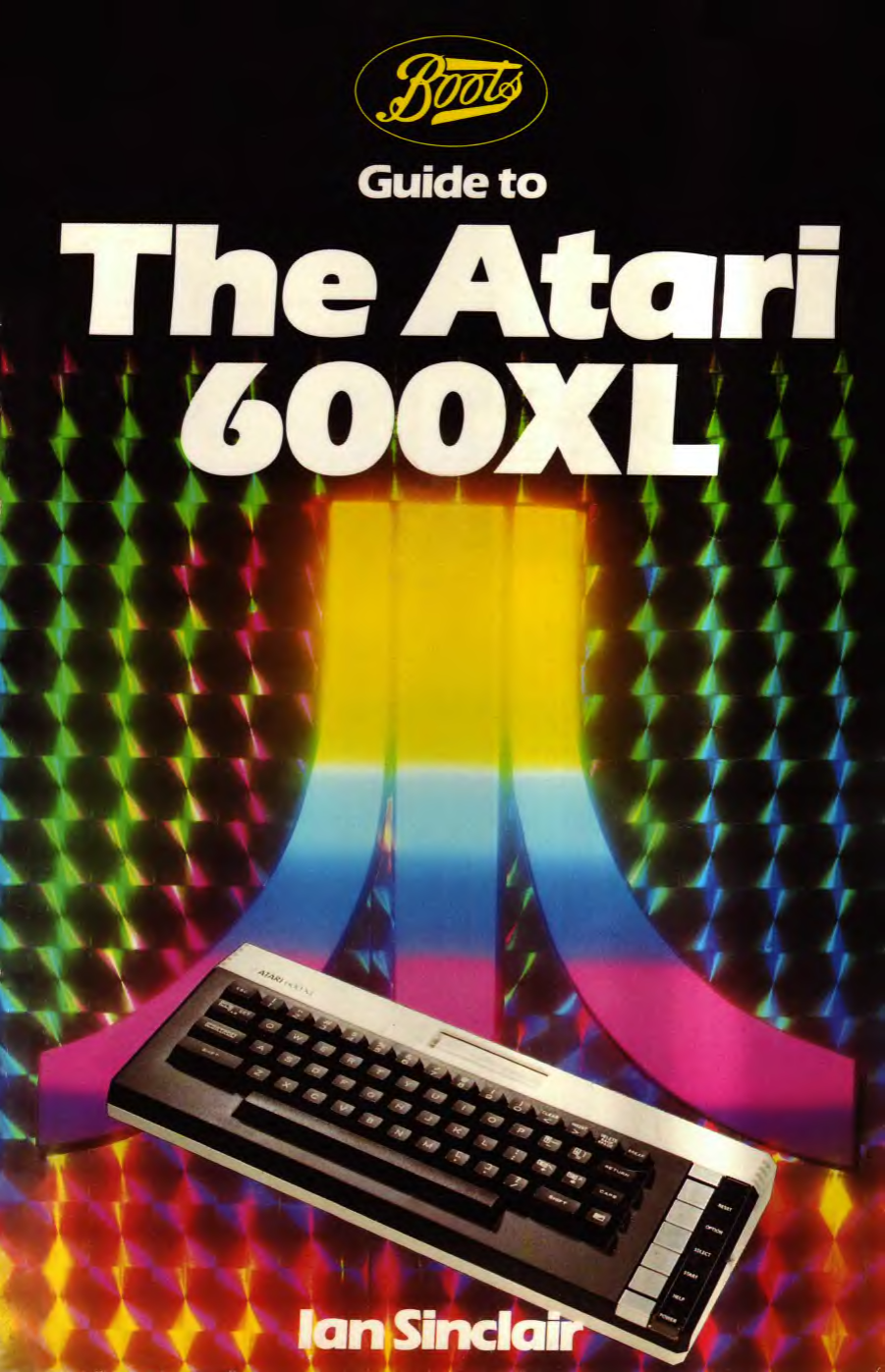
Boots

Guide to

The Atari 600XL



Ian Sinclair

Boots

GUIDE TO THE ATARI 600XL

GUIDE TO THE ATARI 600XL

Ian Sinclair

Published 1984 on behalf of the Boots Company plc,
Nottingham, England by Granada Publishing Limited,
8 Grafton Street, London W1X 3LA

British Library Cataloguing in Publication Data
Sinclair, Ian R.
Boots guide to the Atari 600XL.
1. Atari 600XL (Computer)
I. Title
001.64'04 QA76.8.A/

ISBN 0-246-12370-2

Typeset by V & M Graphics Ltd, Aylesbury, Bucks
Printed and bound in Great Britain by
Mackays of Chatham, Kent

Contents

Preface vi

1 The Computer 1

2 Starting Software 17

3 A Bit of Variety 27

4 Repeating the Process 38

5 String along with Atari 52

6 Special Effects 65

7 High Resolution Graphics 83

8 Sounds of Music 96

9 Do It Yourself! 103

Appendix A: Editing 118

Appendix B: Error Codes 120

Index 121

Preface

The Atari 600XL is a computer that is capable of the most remarkable effects – if you know how to program it. Programming is the best part of computing. Though you can get a lot of pleasure from using programs that other people have written, writing your own programs, presents an irresistible challenge. Running a program that has been yours from concept to typing in gives you a feeling of achievement that practically nothing else can match.

To be able to use the computer effectively, you need to know how to operate it. Programming requires a lot more learning and experience. The aim of this book is to introduce you to the use and the programming of your Atari 600XL. It's only an introduction. A computer as complicated and capable as the Atari can't be completely explained in a single volume, even one three times the size of this book. What I can claim, however, is that this will get you started. By the time you have worked your way through this book, you will be familiar with your Atari, and ready to delve more deeply into its mysteries. You will understand how the computer responds to commands, and you will know how a program is designed. You'll be on the launch-pad, with the main engine ready to fire!

As always, this book owes its appearance to a lot of dedicated work by a lot of dedicated people. I particularly want to thank Richard Miles of Granada Publishing, who has toiled through many weekends to make this book appear in time. My congratulations must also go to Sue Moore and Julian Grover, also at Granada Publishing, and to the long-suffering printers who perform miracles with every one of my books.

Ian Sinclair

Chapter 1
The Computer

A computer is a form of robot, one which doesn't move. What makes it a form of robot? The fact that it can obey your instructions. At the present stage in the development of computers, it won't obey just any instructions, and certainly not if you only speak them. The instructions that the computer will obey must be typed on its keyboard. They must also be in words that the computer understands, words that it has been programmed to recognise. What can it do? To start with, it can produce words, numbers and patterns on the screen. If you use a colour TV connected to your Atari, you can see these results in colours, because the computer can control colours as well as patterns. The Atari can also produce sounds, and you can make it produce the kind of sounds that you want. If you add extra items, called 'peripherals', you can make the computer control a printer, even a colour printer. You can program it so that the screen display or the printer can be affected by the movement of joysticks. You can even use it to control moving robots (called 'turtles'), and to switch electric circuits on and off. In short, a computer can do whatever your ingenious brain can adapt it for. It's the ultimate robot, the one which obeys your every command!

Knowing how to command the machine is all-important, therefore. Certainly, you can buy ready-made programs that will allow you to use your Atari 600XL for games, word processing, accounts, scientific or engineering calculations – a huge range of possibilities. Unless you learn to program it for yourself, though, it's never completely under *your* control. This book is a *starter* guide to the art of programming. I emphasise starter because there's a lot to learn about the Atari 600XL if you want to be able to control it completely. There's far more, in fact, than could fit in one book. All I'm aiming at here, then, is to get you familiar with how to start

programming. From there, you can progress at your own pace, and to your own taste. Getting started is the difficult bit, and that's where you need the help. This is it!

Hardware

The hardware of computing consists of all the bits that you can drop and spill coffee over. For the Atari 600XL, that means the computer itself, the power supply, and the connecting leads. The three-pin plug for the mains lead comes ready-connected to the power supply of your Atari, so there's no electrical work for you to do. All you need to do is to plug all the bits together. Let's take a look at the bits first. The main part is the keyboard. This is the actual computer itself, but to make it work it has to be connected to a power supply, a TV receiver, and also, preferably to the Atari program recorder. The power supply is needed because, unlike washing machines or toasters, computers don't need the full high voltage of the mains to work. What they need is a low voltage supply, the kind you can get from batteries. Batteries don't last long when used with a computer of this class, however, so a special mains adapter is supplied. This is the black box, measuring about 12 cm × 9 cm × 6 cm high, which is provided with two cables. One of these cables is a thinner one, ending in a small seven-pin plug – the type called a DIN plug (Fig. 1.1). This plug is inserted into the

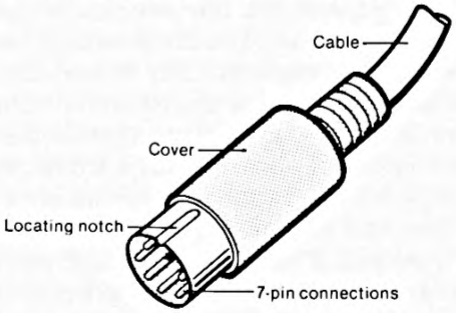


Fig. 1.1. The power cable to your Atari. This comes from the power supply box, and plugs into the PWR.IN socket at the back of the computer.

power socket of the Atari. This socket is labelled PWR.IN and is next to the ON/OFF switch at the back of the computer, on the left-hand side as you look at the keyboard. The DIN plug can fit only one way round – look for the locating notch in the plug and in the socket. The

other cable of the power supply ends in a 3-pin plug of the type that is used in practically all homes in Britain. If your home doesn't have sockets that take this kind of plug, it's time your house was rewired! In the meantime, you can buy an adapter, or have an electrician put on a different type of plug.

An important point to note here is the use of the ON/OFF switch of the Atari. All this does is to switch the low voltages that the computer uses. It won't switch off the high mains voltage, so always remove the mains plug from its socket when you have finished a session of computing. Don't remove the small DIN plug any more than is absolutely necessary (like when you are shifting the computer from one place to another). If this plug loosens and starts to make uncertain contact, your computer simply won't work reliably. Whenever the power supply is interrupted, even if only for a fraction of a second, the computer loses all of the instructions that you had commanded it to carry out. If these instructions took you an hour to enter, you won't feel too pleased about that!

TV time

With that done, you are almost ready to work some Atari 600XL magic, but you need the use of a TV receiver. A computer is a device which is arranged so as to send signals to a TV receiver, and unless you connect a TV receiver to the Atari 600XL you won't be able to see what the computer is doing. It will still compute for you just as well, but you won't see what is going on. The Atari 600XL comes with its TV cable ready to attach, with an aerial plug at one end of the lead, and a different type of plug (a phono plug) at the other end. The two different plugs are illustrated in Fig. 1.2. You could, of course, simply

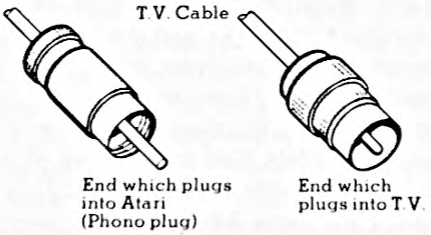


Fig. 1.2. The TV connector cable. The plugs at the two ends are different, and you can't interchange them.

plug this lead into the TV receiver, but a better option is to use the type of 2-to-1 adapter that is illustrated in Fig. 1.3. This allows you to keep an aerial cable plugged in, and to connect or disconnect the Atari 600XL as you wish without disturbing the TV receiver. It's

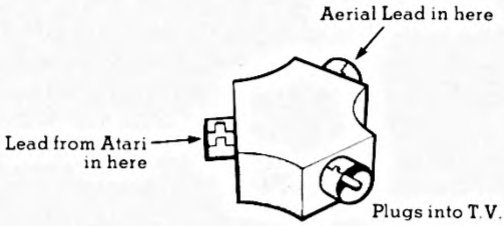


Fig. 1.3. A two-way TV adaptor. This can save a lot of the wear and tear on sockets, because it allows the TV to be connected to an aerial as well as to the Atari.

useful if you have to share a colour TV with the family. It also saves wear on the aerial connector of the TV receiver itself. If you have a TV that you can reserve for use with the Atari then you won't need this device. The TV that you use to display the computer's signals need not be a colour receiver, not to start with at least. The skills of programming an Atari 600XL do not require you to see the results in colour until you come to the colour instructions in Chapter 6. If you use a black/white receiver, such as the little Ferguson portable which has served me so well, you will see the Atari colours as shades of grey.

Socket to me

Now before you plug in everything in sight and switch on, it's a good idea to see how many mains sockets you have around. When you are in full control of your Atari you will need three mains sockets. Two of these will be for the Atari 600XL and the TV receiver, but you will need one more for a program recorder. Most houses have desperately few sockets fitted, so you will find it worthwhile to buy or make up an extension lead that consists of a three- or four-way socket strip with a cable and a plug (Fig. 1.4). This avoids a lot of what the famous advert calls 'spaghetti hanging out of the back'. Don't rely on the old-fashioned type of three-way adapter – they never produce really reliable contacts. As I have said, when a computer is disconnected from its power supply, it instantly loses anything that was in its

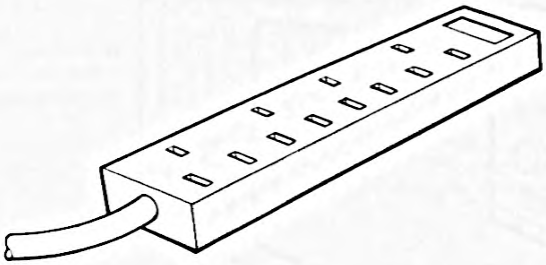


Fig. 1.4. A four-way socket strip which avoids the use of the old-style adapters.

memory. Always start a computing session by plugging in the mains plug, then switching on the Atari 600XL. End by switching off the Atari 600XL then, finally, unplugging the mains plug.

The next step, then, is to switch on the TV receiver and the computer. The signals have to be transmitted, using a miniature transmitter that is called a 'modulator'. This is because most TV receivers cannot be safely connected to anything except by the aerial lead. The TV receiver has to be tuned to these signals from the Atari 600XL. Unless you have been using a video cassette recorder, and the TV has a tuning button that is marked VCR it's unlikely that you will be able to get the Atari tuning signal to appear on the screen of the TV simply by pressing tuning buttons. The next step, then, is to tune the TV to the computer's signals.

Figure 1.5 shows the three main methods that are used for tuning TV receivers in this country. The simplest type is the dial tuning system that is illustrated in Fig. 1.5(a). This is the type of tuning system that you find on black/white portables, and to get the Atari 600XL's signal on the screen, you only have to turn the dial. If the dial is marked with numbers, then you should look for the signal somewhere between numbers 30 and 40. If the dial isn't marked, which is unusual, then start with the dial turned fully anticlockwise as far as it will go, and slowly turn it clockwise until you see the Atari 600XL signal appear.

What you are looking for, if the Atari 600XL hasn't been touched since you switched it on, is the word READY at the top left-hand corner of the screen, and a light square under the R of READY. This square is called 'the cursor', and it indicates where a letter will appear on the screen if you press a letter key. On a B/W receiver, you will see only black and white, but a colour receiver will show you that the

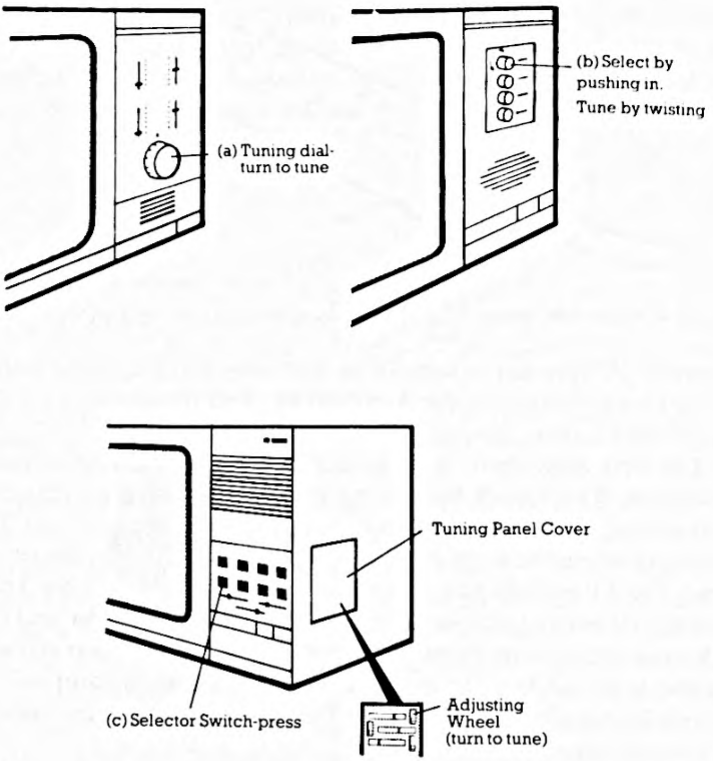


Fig. 1.5. TV tuning controls. (a) Single dial, as used on black and white portables, (b) four-button type, (c) the more modern touch-pad or miniature switch type.

letters are light blue on a dark-blue background. You have to adjust the tuning dial until you can see the word READY. When you can see the word, turn the dial carefully, turning slightly in each direction until you find a setting in which the word is really clear. It helps if you turn up the volume control of the TV receiver as well. When the signal is correctly tuned in, the amount of rushing noise that you hear from the loudspeaker will be a minimum. On a TV receiver, particularly a colour TV, the word may never be especially clear, but get it steady at least and as clear as possible. Figure 1.6 illustrates some faults that are caused by incorrect tuning.

The older types of colour and B/W TV receivers used mechanical push-buttons (Fig. 1.5(b)) which engage with a loud clonk when you

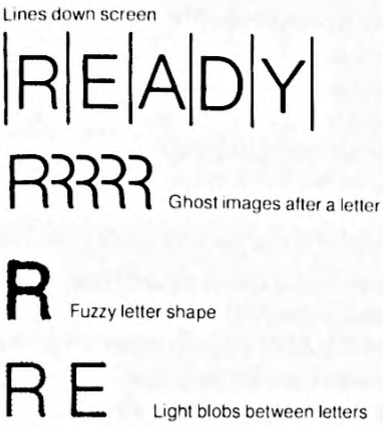


Fig. 1.6. The sort of faults that can be caused by incorrect tuning. If you turn up the volume control, you can also tune for minimum sound.

push them. There are usually four of these buttons, and you'll need to use a spare one which for most of us means the fourth one. Push this one fully in. Tuning is now carried out by rotating this button. Try rotating anticlockwise first of all, and don't be surprised by how many times you can turn the button before it comes to a stop. If you tune to the Atari 600XL's signal during this time, you'll see the same sign – and the message on the screen, and the sudden hush from the loudspeaker. If you've turned the button all the way anticlockwise and not seen the tuning signal, then you'll have to turn it in the opposite direction, clockwise, until you do. If you can't find the Atari 600XL signal at any setting, check the TV using an aerial in case there is something wrong with the tuning of the TV.

Modern TV receivers are equipped with touch pads or very small push-buttons for selecting transmissions. These are used for selection only, not for tuning. The tuning is carried out by a set of miniature knobs or wheels that are located behind a panel which may be at the side or at the front of the receiver (Fig. 1.5(c)). The buttons or touch pads are usually numbered, and corresponding numbers are marked on the tuning wheels or knobs. Use the highest number that is available (usually 6 or 12), press the pad or button for this number, and then find the knob or wheel which also carries this number. Tuning is carried out by turning this knob or wheel. Once again, you are looking for a clear picture on the screen. On this type of receiver,

the picture is usually 'fine-tuned' automatically when you put the cover back on the tuning panel, so don't leave this cover off. If you do, the receiver's circuits that keep it in tune can't operate, and you will find that the tuning alters, so that you have to keep retuning.

Mystery and mastery

Once you have achieved a tuned signal from your Atari 600XL, the business of mastering the Atari 600XL magic begins. To start with, you have the word READY shining at you from the top of the screen. It's important to note that nothing that you can do by pressing keys on the keyboard can possibly damage the Atari 600XL – the worst you can do is to lose a program that was stored in the memory. You can, however, damage the Atari 600XL by spilling coffee all over it, dropping it, or connecting it up to other circuits while the power is switched on. Pressing keys at random can, however, cause the computer to look as if it has 'seized up', refusing to do anything. You can always escape from these tantrums in two ways. One way is to press the key that is marked BREAK. This key is on the top row of keys, at the right-hand side of the keyboard. It should be your first resort if you think the computer has 'locked up'. The second resort is the uppermost silver key at the right-hand side of the machine – it's marked RESET. Pressing this will cause the picture to blacken, clear, and display READY at its top left-hand corner again. Your ultimate cure is to switch off – but that's rather drastic.

One final point about this. If you leave the Atari switched on for a long time, but don't touch the keys, you will see the colours change at intervals. There's nothing wrong with this. It's simply a way of ensuring that a colour TV displays all of its colours for equal times. This prevents uneven wear! The display will return to normal when you press any key. It's just another of the thoughtful features built into your Atari 600XL.

It's time now to look at the keyboard, because the keyboard is the way that you pass instructions to the Atari 600XL. Most of the Atari keys are arranged rather like typewriter keys. The arrangement of letters and numbers is the same as that of a typewriter and if you've ever used a typewriter, particularly an electric typewriter, then you should be able to find your way round the keyboard of the Atari 600XL pretty quickly.

There's one very noticeable difference, though. When you use a typewriter, pressing a letter key produces a small letter (lower-case), and pressing the SHIFT key along with a letter key produces a capital letter (upper-case). On the Atari, you will see upper-case (capitals) produced from letter keys whether you press the SHIFT key or not. That's because instructions have to be typed in upper-case. If you want to use the keyboard of the Atari 600XL like the keyboard of an ordinary typewriter, you have to press the CAPS key which is at the right-hand side of the keyboard. After this, any letter key pressed along with SHIFT will produce an upper-case letter. A key pressed alone will produce lower-case. To get back to the upper-case only condition, which you should use for programming, press the CAPS key once again. The symbols that you see on the upper part of some keys are obtained by pressing the SHIFT key along with that letter/number key. All of the keys will repeat their action if you hold them down.

As well as the ordinary typewriter keys, there are ten special keys which are not found on any typewriter. These are the BREAK key, the RETURN key, the five silver keys at the extreme right-hand side, and the INVERSE VIDEO, ESCAPE and CONTROL keys. The most important of these special keys, however, as far as we are concerned at the moment, is the key that is marked RETURN. This is in the position of the 'carriage return' key of an electric typewriter, but its action is *not* the same in all respects. Pressing the RETURN key is a signal to the computer that you have completed typing an instruction and that you now want the computer to obey it.

If you are accustomed to using an electric typewriter, you will have to change some of your habits as far as this key is concerned. During the use of a typewriter, you would press the 'carriage return' key each time you wanted to select a new line, with typing starting at the left-hand side of the new line. The RETURN key of the computer does rather more than this. If the material that you are typing into the Atari 600XL takes more than one line on the screen, the machine will automatically select the next screen line for you. The RETURN key must not be used for this purpose. The RETURN key is used only when you want the machine to carry out a command or store an instruction, not simply when you want to use a new line. It will always provide a new line for you, however, and select a position at the left-hand side. The position where a letter or other character will appear when you press a key is indicated by a flashing square on the screen.

This flashing square is called the cursor, and it acts as a sort of signpost for you, as we'll see later.

Testing time for Atari

One of the unique features of the Atari 600XL, which distinguishes it from all other computers, is its self-testing. This makes use of a number of the special silver keys, and it also gives you a very useful idea of what the computer can do. There are two ways of getting the computer to swing into its self-test routine. One of them is used at the time when you switch on. If you hold down the OPTION key (second from the top, silver keys) at the instant when you switch the computer on, it will go into the self-test routine. If you have already switched the computer on, you don't have to switch off again. Just type BYE (yes, BYE, just as it looks), and then press the RETURN key. Either of these actions will present you with the SELF-TEST MENU - and will give you a flavour of what computing is about!

When the menu appears, you will see that one of the phrases is flashing. The phrases are MEMORY, AUDIO-VISUAL, KEYBOARD, and ALL TESTS. The one that is flashing is the one that will be selected. To alter it, press the SELECT key. This will shift the selection. If the MEMORY phrase was flashing before, then pressing SELECT once will make AUDIO-VISUAL flash. Press again, and KEYBOARD will flash. Press again, and ALL TESTS will flash. There aren't any more options, so pressing SELECT yet again returns you to MEMORY. That's as good a place to start as any other. With MEMORY flashing, then, press the START key, just above the HELP key.

The memory test shows on the screen two bars, under the word ROM, and a set of 16 small squares under the word RAM. The bars and the squares will change colour as you watch them. Normally, they are dark green, but at intervals they will light up in light blue. This indicates that all is well with your computer. If you see the bars or the squares turn red or purple, then there are problems, and you should contact your Atari dealer, describing what happened. Computers are reliable, so it's unlikely that you'll ever have to worry about this. It's comforting to know, however, that you can check the action of your computer so simply for yourself.

Now press the HELP key, and you'll find yourself looking at the menu again. Select a new check this time, the AUDIO-VISUAL. This

is a check of the screen and the sound output of the Atari, so you need to have your TV correctly adjusted, and the volume control turned up. The sound signals of your Atari are transmitted to the TV with the picture signals, so that you have complete control over them with the volume control of the TV. When you press START with this option selected, the computer draws a set of musical lines and spaces on the screen. It then prints the notes and plays them! If the pattern and the sound are OK, then this part of the computer's system is also working perfectly.

The last option is KEYBOARD. Press HELP to get back to the menu, and select the KEYBOARD option. This prints on to your screen a picture of the keyboard keys, yellow on green if you are using a colour TV. When you press a key on the keyboard, the corresponding key on the screen should flash, and a note should sound. All of the letter and number keys will indicate in this way, but the SHIFT and CONTROL keys will flash only when another key is pressed at the same time. The BREAK key has no effect, and pressing HELP will return you to the menu. Pressing RESET will return you to the ordinary screen display, with READY showing. Note when you use this one, that some of the symbols on the screen do not look the same as the symbols on the keys. In addition, the broken row of 'keys' that is displayed on the top of the screen refers to the silver keys, and also to other tests which are not used on the 600XL.

If you select the ALL TESTS option, then the computer will cycle through all of the tests in turn until you press HELP (to get to the menu again) or RESET (to get back to ordinary computer use).

The cartridge slot

At the top of your keyboard unit there is a slot which is covered by two spring-loaded flaps. This slot is for programs which are recorded in cartridge form. To use a cartridge program, switch off the computer, and insert the cartridge, with the label facing you, into the slot. Then switch on. You may have to use RESET to get the program going, but from then on the instructions will be contained in the program itself. It's easy! Programs of this sort often make use of the silver keys. Note, however, that when you are programming for yourself these keys have no effect, apart from the RESET key.

The other control keys

In addition to the silver keys and the RETURN key, the keyboard of your Atari contains some other special-action keys. You will discover for yourself, as you progress with this book, how they are used. At this point, though, it's useful to have a note of what each one does, just for reference.

The BREAK key is used to interrupt the computer when it's working. Pressing this key will stop an ordinary program (it doesn't always stop a program that is on a cartridge), so allowing you to make changes. The ESC (ESCAPE) key is used along with other keys to obtain special characters, and we'll look at this use later. The CONTROL key is also always used along with other keys. We'll look at its use in editing in Appendix A. It can also be used with the number keys, 1, 2 and 3. Of these, pressing CONTROL and 2 together will cause a buzzer-like sound. CONTROL 1 will stop the keys from having any effect on the screen (and CONTROL 1 pressed again will restore normal action). CONTROL 3 is for special purposes.

Finally, there is the INVERSE VIDEO key. It's not marked INVERSE VIDEO, which makes life rather confusing. Instead, it's marked with a rectangle, diagonally split with one part white and the other dark. On the old Atari computers, this key was marked with the Atari symbol, and in some programs you may find it referred to as the 'Atari key'. Pressing this key will cause everything to print on the screen in 'inverse video', that is, dark on light. If you type program commands in this way, they won't be obeyed, so the key has to be kept for special effects in printed messages.

Saving it!

You can obtain a lot of enjoyment from a computer system that consists only of the machine and a TV receiver. Each time that you switch the machine off, however, all the program and other information that has been stored in the memory of the computer will be lost. Since it might take several hours to enter a program into the machine by typing instructions on the keyboard, this waste just has to be avoided. We avoid the loss of programs by recording them on tape. Before you tackle the rest of this book, then, it's important to check now that you can record and replay programs.

Only the Atari program recorder is usable with Atari computers. For the 600XL you should have the Type 1010 program recorder, but

I have used the older Type 410 with success. The advantage of using a special recorder is that it's designed for the job. Ordinary cassette recorders that other computers use were never designed for recording computer programs, and many users experience trouble with them. You can't adapt an ordinary cassette recorder to use successfully with the Atari, so it's as well to buy your program recorder at the same time as you buy the computer.

Start work by switching everything off. Now take a look at the cable that is used to connect the Atari 600XL to its program recorder. This is permanently connected to the program recorder at one end. The other end is fitted with a special 13-pin plug (see Fig. 1.7). This

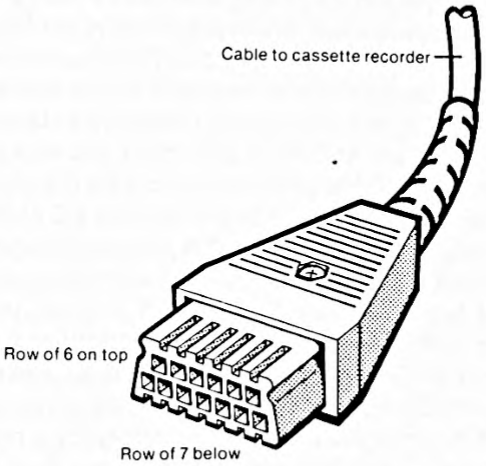


Fig. 1.7. The plug at the end of the program recorder cable. This can only go in one way round. Don't force it!

plug will go in only one way round. On the Atari 600XL keyboard, the plug of this cable is inserted into the socket that is marked PERIPHERAL, at the right-hand side of the rear of the keyboard. The plug is a tight fit - ease it into place without using undue force.

Once you have made this connection to the recorder, the program recorder is ready for use. The recorder also needs a power supply, and the supply that comes with it is fitted with a jack plug on one cable and a 3-pin mains plug on the other. The usual type, in fact, contains the power supply as part of the 3-pin plug. The jack plug is completely unlike the plug that is used to connect power to the keyboard, so you can't mix them up. The next thing that you have to sort out is a supply

of blank cassettes. There's nothing wrong with using reputable brands of C90 length cassettes (ordinary 'ferric' tape, not the Hi-Fi CrO_2 type), but you'll find that the short lengths of tape that are sold as C5, C10 or C15 in computer shops and in many chainstores are much more useful.

Put a fresh cassette into the machine, with the 1 or A side uppermost. The first part of the cassette consists of a 'leader', which is plain, not recording, tape. This has to be wound on before you can record. If your recorder has a tape counter, reset the counter to zero, and then fast-wind the cassette to a count of 5. If there is no tape counter, take the cassette out and insert the body of a BiC pen into the centre of the empty reel. Turn the pen so that the tape winds on to the reel, and keep turning until you see the brown recording tape replace the clear or brightly-coloured leaders.

Now before you can make a recording to test the system, you need a program to record, and this involves some typing. This is easy if you have just switched the Atari 600XL on, but, if you have been pressing keys at random then it's a good idea to clear the memory. Do this by typing the word NEW, and then pressing the RETURN key. This does not produce any obvious effect, apart from the word READY on the screen. You can wipe the screen clear by pressing the CLEAR and SHIFT keys together.

Type the number 1Ø (1 and then Ø), and then type the word REM just following the 1Ø. Check that this looks correct, and then press the RETURN key. The effect of this is to place the instruction line 1Ø REM into the memory of the machine. Now type the rest of the lines, as illustrated in Fig. 1.8, remembering to press the RETURN key

```
10 REM
20 REM
30 REM
40 REM
```

Fig. 1.8. A program for testing the cassette recording and replaying actions.

after you have completed typing each line. The numbers are called *line numbers*, and they are there for two reasons. One is to remind the computer that this is a program, the other is to guide it, because the computer will normally carry out instructions in the same order as the line numbers. Following a line number, the computer must have an instruction word, and REM is such a word. It's not an exciting instruction, just one that tells the computer to ignore whatever follows it!

Check that your program looks on the screen like the printed version in Fig. 1.8, and make sure that the cassette recorder is ready. Now type the word CSAVE. CSAVE is the instruction to the computer meaning that you want to save (record) a program on a cassette. Now press the RETURN key, and the computer will honk twice at you. Remember that the sound comes from the TV loudspeaker, so if you have the volume turned down you won't hear it! Now press the PLAY and RECORD keys on the recorder. Press them firmly so that they lock in place. Now press the RETURN key again, and you will hear the recorder motor start and see the reels of the cassette turning. A high-pitched sound comes from the loudspeaker of the TV and, after a while, this sound becomes a rough note. This is the set of notes that is a coded version of the program. After a time, the word READY reappears on the screen, and the motor of the cassette recorder stops. This lets you know that the program has been recorded, assuming that you did everything correctly.

Now comes the crunch. You have to be sure that the recording was O.K. Wind back the tape, using the rewind key of the recorder, and type NEW, then press RETURN. This command will cause the Atari to erase the program. You can check this by typing LIST (then press RETURN). Nothing is listed on the screen, because there is no program to list now, until you play the program back. Type CLOAD (Cassette LOAD), and press RETURN. The computer will honk at you, just once this time. Now press the PLAY key of the recorder, and then the RETURN key of the computer. When the program is loaded into the computer, the READY message will appear again. The program recorder motor will stop, and you should press the STOP key on the recorder. If you keep the PLAY key of the recorder pressed down for long periods when the motor is not running, you can damage the recorder. Type LIST now, press RETURN, and you should see your program on the screen. This proves that your cassette recording is O.K., and correct recordings are being made.

This action should be completely trouble-free. You don't have to make any of the endless fiddling adjustments that owners of other computers are tormented with. Once you can reliably save programs on tape and reload them, you can confidently start computing. When you have spent an hour or more typing a program on to the keyboard, it's good to know that a few minutes more work will save your effort on tape so that you won't have to type it again. Note that the older

type of Atari program recorder will work perfectly with the 600XL, but the new recorder has provision for recording stereo sound as well as programs.

Finally, a warning. At the back of your Atari, you will see a covered slot which is labelled PARALLEL BUS. Unless you are going to plug anything into this, keep the cover on, and make sure that no liquid or metal can find its way in. If there are small fingers about, glue the cover down until you have an expansion unit to plug into it.

Chapter 2
Starting Software

Chapter 1 will have broken you in to the idea that the Atari 600XL, like practically all computers, takes its orders from you when you type them on the keyboard. You will also have found that an order is obeyed when the RETURN key is pressed. You have used the command NEW which clears out a program from the memory; and LIST which prints your program instructions on to the screen. Now there are two ways in which you can use a computer. One way is called *direct mode*. 'Direct mode' means that you type a command, press RETURN, and the command is carried out at once. This can be useful, but the more important way of using a computer is in what is called *program mode*. In program mode the computer is issued with a set of instructions, with a guide to the order in which they are to be carried out. A set of instructions like this is called a *program*.

The difference is important, because the instructions of a program can be repeated as many times as you like with very little effort on your part. A direct command, by contrast, will be repeated only if you type the whole command again, and then press RETURN.

Let's take a look at the difference. If you want the computer to carry out the direct command to add two numbers, 1.6 and 3.2, then you have to type:

PRINT 1.6 + 3.2 (and then press RETURN)

You have to start with PRINT because a computer is a dumb machine, and it obeys only a few set instructions. Unless you use the word PRINT, the computer has no way of telling that what you want is to see the answer on the screen. It doesn't recognise instructions like 'TELL ME' or 'WHAT IS', only a few words that we call its 'reserved words' or 'instruction words'. PRINT is one of these words. When you press RETURN after typing PRINT 1.6 + 3.2, the screen shows

the answer, 4.8, on the next line of the screen below, and the message 'READY' just under it. The word READY is what we call a 'prompt'. It's the computer's way of telling you that it has finished the task you commanded it to do, and is waiting for another. Once this command has been carried out, however, it's finished.

A program does not work in the same way. A program is typed in, but the instructions of the program are not carried out when you press RETURN. Instead, the instructions are stored in the memory, ready to be carried out as and when you want. The computer needs some way of recognising the difference between your direct commands and your program instructions. This is done by starting each program instruction with a positive whole number which is called a 'line number'. This is why you can't expect the computer to understand an instruction like 5.6 + 3 = ; it would expect the 5 to be a line number, and having a fraction just doesn't make sense to it. That's why, if you try to type a line like that, you'll see, when you press RETURN, the message:

6 ERROR + 3 =

(with the + sign printed black-on-white). The rules that you have learned so far, plus more to come, form a 'programming language'. This particular programming language is called BASIC (Beginner's All-purpose Symbolic Instruction Code). It started out as a language for teaching people about computers, but has now become one of the most popular computer languages for all sorts of uses.

Let's start programming, then, with the arithmetic actions of add, subtract, multiply and divide. Why start with these? Computers aren't used all that much for calculation, but it's useful to be able to carry out calculations now and again. Figure 2.1 shows a four-line program which will print some arithmetic results.

```
10 PRINT "2.6+3.3= ";2.6+3.3
20 PRINT "1.5*4.7= ";1.5*4.7
30 PRINT "7.4/2.2= ";7.4/2.2
40 PRINT "8.5-2.4= ";8.5-2.4
```

Fig. 2.1. A four-line arithmetic program.

Take a close look at this program, because there's a lot to get used to in these four lines. To start with, the line numbers are 10,20,30,40 rather than 1,2,3,4. This is to allow space for second thoughts. If you decide that you want to have another instruction between line 10 and line 20, then you can type the line number 15, or 11 or 12 or any other

whole number between 1Ø and 2Ø, and follow it with your new instruction. Even though you have entered this line out of order, the computer will automatically place it in order between lines 1Ø and 2Ø. If you number your lines 1,2,3 then there's no room for these second thoughts.

The next thing to notice is the shape of the number zero when you see it on the screen. It has a diagonal slash across it (Ø). This is to distinguish it from the letter O. The computer simply won't accept the zero in place of O, nor the O in place of zero, and the slashing makes this difference more obvious to you, so that you are less likely to make mistakes. The zero key on the keyboard does not show a slash through the zero, however. Some magazines, unfortunately, reprint computer programs with the slashmarks removed, so that it's very easy to make mistakes.

Now to more important points. The star or asterisk symbol in line 2Ø is the symbol that the Atari 600XL uses as a multiply sign. We can't use the × that you might normally use for writing multiplication because × is a letter. There's no simple way that a machine can tell when you want to use × as a letter and when you want to use it as a multiply sign. The machine therefore takes the easy way out – it always uses the asterisk (star) sign to mean multiply. There's no divide sign on the keyboard either, so the Atari 600XL, like all other small computers, uses the backslash (/) sign in its place. The backslash is on the same key as the question mark. Don't confuse it with the other slash sign (\) on the + key.

So far, so good. The program is entered by typing it, just as you see it. You don't need to leave any space between the line number and the P of PRINT, because the Atari 600XL will put one in for you. You don't even have to type the whole word PRINT. You can use the question mark (?) in place of the word PRINT! When you later ask the computer to list the lines of the program on the screen, each of these question marks will have been turned into the word PRINT. Not such a dumb machine, after all, maybe! The change from ? to PRINT, and the gap between the line number and the PRINT is not visible at the time when you type the program, but you will see it later, when you LIST. You will have to press the RETURN key when you have completed each instruction line, before you type the next line number. You should end up with the program looking as it does in the illustration. When you have entered the program by typing it, it's stored in the memory of the computer in the form of a set of code

numbers. There are two things that you need to be sure of now. One is how to check that the program is actually in the memory, the other is how to make the machine carry out the instructions of the program.

The first part is dealt with using the command LIST that you know already. You can press the CLEAR key (remember to use the SHIFT key as well) to wipe the screen first if you like, then type LIST and press the RETURN key. When you press the RETURN key, and not until, your program will be listed on the screen. You will then see how the computer has printed the items of the program on the screen, with spaces between the line numbers and the instructions. Any time that you are uncertain of what you may have in the memory of the Atari, just LIST it like this. If you have a long program in the memory, you will see the lines start to move up the screen, so that the first lines of the program are no longer visible. To stop this, you can list a few lines at a time. For example, if you type LIST1Ø,1ØØ, the screen will display line 1Ø and all the lines up to and including line 1ØØ.

To make the program operate, you need another command, RUN. Type RUN, then press the RETURN key, and you will see the instructions carried out. In each line, some of the typing is enclosed between quotes (inverted commas) and some is not. Can you see how very differently the computer has treated the instructions? Whatever was enclosed between quotes has been printed exactly as you typed it. Whatever was not between quotes is worked out, so the first line, for example, gives the unsurprising result:

```
2.6 + 3.3 = 5.9
```

Now there's nothing automatic about this. The computer can't think and it doesn't know that 2.6 + 3.3 = 5.9. If you type a new line:

```
15 PRINT "2.6 + 3.3 =  ";2+2
```

then you'll get the daft reply, when you RUN this, of:

```
2.6 + 3.3 = 4
```

The computer does as it's told and that's what you told it to do. Why should computers take over the world when you can command them to do things as daft as this?

This is a good point also to take notice of something else. The line 15 that you added has been fitted into place between lines 1Ø and 2Ø – LIST if you don't believe it. No matter in what order you type the lines of your program, the computer will sort them into order of rising

line number for you. Note also that the spaces in the program of Fig. 2.1 between the = and the " are useful – just see what happens if you miss them out!

With all of this accumulated wisdom behind us, we can now start to look at some other printing actions. PRINT, as far as the Atari 600XL is concerned, always means print on to the TV screen. For activating a paper printer, like the Atari 1025 80-column printer, you need a different instruction, LPRINT. You must not use this instruction in a program unless you have a printer connected and switched on. If you accidentally type LPRINT, edit it out of your program before you RUN it. If you do run a program with an LPRINT in it, and no printer connected, then you'll have to press the RESET key to get back to normal.

Now try the program in Fig. 2.2. You can try typing the lines in any order that you like, to establish the point that they will be in line

```
10 ? "THIS IS ATARI 600XL"
20 ? "YOUR ENTRY";
30 ? " TO THE WORLD OF COMPUTING"
40 ? "IN COLOUR"
```

Fig. 2.2. Using the PRINT instruction to place words on the screen. The ? sign has been used in place of PRINT. Notice the effect of the semicolons.

number order when you list the program. Note, by the way, how the ? sign has been used in place of PRINT. This is a useful dodge which can save a lot of typing. As you gain experience, you will find that a lot of the instruction words of the Atari can be shortened so as to make life easier for you. When you RUN the program, the words THIS IS ATARI 6ØØXL appear on one line, and the words YOUR ENTRY TO THE WORLD OF COMPUTING on the next line. This is because the instruction PRINT doesn't just mean 'print on the screen'. It also means 'take a new line and start at the left-hand side'! Whatever you have put between quotes, however, will be printed just as you type it, upper-case, lower-case or inverted video, the lot.

Now this automatic new line isn't always convenient, and we can change the action by using punctuation marks that we call 'print modifiers'. In line 2Ø, for example, the semicolon at the end of the line will cause printing to continue on the same line. You have to be careful how you do this, because you will jam words together if you don't leave a space, and you also have to watch how words are split up at the end of a line. The Atari 600XL allows you 38 characters in each printed line. This can be changed to a maximum of 40, or to smaller

numbers if you like, but usually there's no point in changing it. For the moment, though, we have another printing trick to learn.

Start this time by acquiring a good habit. Type NEW and then press the RETURN key. This clears the old program out. If you don't do this, there's a chance that you will find lines of old programs getting in the way of new ones. Each time you enter a line, you delete any line that had the same line number in an older program, but if there is a line number that you don't use in the new program it will remain stored. For example, suppose you have a program which uses line numbers 1Ø, 2Ø, 3Ø and 4Ø. Now if you type a new program, with lines 1Ø, 2Ø and 3Ø, then these new lines replace the old ones. The old line 4Ø is still there, however, lurking in the memory, and it will join on to the new program. Take another example. Suppose that your new program used lines 5, 15, 25, and 35. After you typed these lines, you would find that you had a program with lines 5, 1Ø, 15, 2Ø, 25, 3Ø, 35, and 4Ø! If you type NEW, then press RETURN, you erase all the old lines, so they can't cause you any trouble. Make sure, before you type NEW, though, that you really want to remove the old program. If you are ever likely to need it again, then record it on tape or disk.

Rows and columns

Neat printing is a matter of arranging your words and numbers into rows and columns, so we'll take a closer look at this particular art now. Figure 2.3 shows one way of arranging columns. Line 1Ø is a PRINT instruction that acts on the numbers 1,2,3 and 4. When these

```
10 ? 1,2,3,4
20 ? 1,2,3,4,5
30 ? "ONE","TWO","THREE","FOUR"
40 ? "THIS IS TOO LONG!","TWO","THREE"
,"FOUR"
```

Fig. 2.3 Using commas to create columns on the screen display.

appear on the screen, though, they appear spaced out just as if the screen had been divided into four columns. The mark which causes this effect is the comma, and the action is completely automatic. You can't get more than four columns. Anything that you try to get into a fifth column will actually appear on the first column of the next line down. The action works for words as well as for numbers, as lines 3Ø and 4Ø illustrate. When words are being printed in this way, though, you have to remember that the commas must be placed outside the

quotes. Any commas that are placed inside the quotes will be printed just as they are and won't cause any spacing effect. You will also find that if you attempt to put into columns something that is too large to fit, the long phrases will spill over to the next column, and the next item to be printed will be at the start of the next column along.

Commas are useful when we want a simple way of creating four columns. Another method of placing words along a line exists, however, and is illustrated in Fig. 2.4. It uses the TAB key of the Atari

```
1Ø ?"START"
2Ø ?"►FIRST TAB"
3Ø ?"►►SECOND TAB"
4Ø ?"►►►THIRD TAB"
```

Note: the ► mark is produced on the screen by pressing the TAB key.

Fig. 2.4. Using the TAB key. This example has been typeset, because my printer does not reproduce the mark that the TAB key makes.

600XL, which is placed next to the 'Q' key, at the left-hand side of the top row of letter keys. The TAB key of the Atari works just like the TAB key of a typewriter. This doesn't mean very much to you if you don't use a typewriter, so let's take a closer look. When you press the TAB key, what you see on the screen is the cursor moving to a different position. To put this instruction for movement into a program, however, you have to press the ESC key, release it, and then press TAB. This will place in the program line a triangular mark, a sort of arrowhead pointing right. What happens when the program runs, however, is that each TAB mark causes a movement of eight places to the right, measured from the start of the line. The first TAB does not appear to be as much as this, because the normal line starts two spaces in. Figure 2.5 shows where these TAB positions are normally. If you try to TAB backwards, nothing happens. For

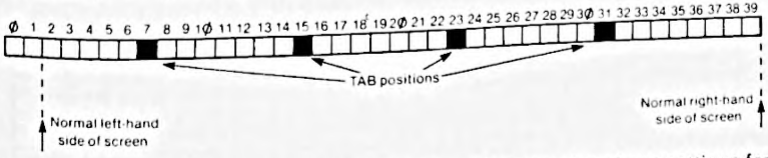


Fig. 2.5. The standard settings for the TAB key. You can alter these settings for yourself!

example, if you have the instruction: PRINT "LEGENDARY... and you follow this with a TAB, and then ATARI", you might expect that

the TAB will cause the A of ATARI to be placed on top of the Y of LEGENDARY, the eighth place along. It's smarter than this, though. If you've moved past a TAB position, it won't move back to it. Instead it will move to the next TAB position along to the right. The only problem with the use of TAB, as far as I am concerned, is that my printer does not print the TAB mark, and I can't demonstrate the effect so easily to you.

The use of tabulation is made a lot more flexible by the fact that you don't have to use the TAB positions that the computer provides. You can place the TAB stops where you want along a line. To do this, you have to start by clearing all the existing TAB stops. Press the TAB key to get to the first TAB position. Then press CONTROL and TAB together. This clears the TAB. Press TAB to move to the next position, and press CONTROL and TAB together. This clears the second TAB stop. Continue this way until all the TAB stops have been cleared. You can check this easily – the cursor will move down the screen instead of across when you press the TAB key. Now you can place the TABs where you want them. Position the cursor, using the spacebar or the backspace key. Press SHIFT and TAB together. This will place a TAB where you had the cursor. You can have as many TAB stops as you like, up to forty in a line!

Meantime, there's another very important print modifier to look at. The instruction POSITION can be used to allow text (numbers,

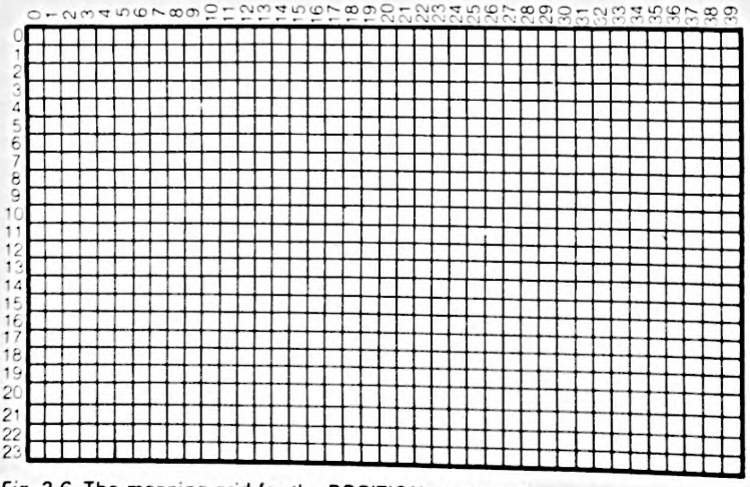


Fig. 2.6. The mapping grid for the POSITION command.

letters, words) to be placed anywhere on the screen. For the purpose of using POSITION, we imagine the screen divided into a grid of 40 divisions across and 24 lines down, as Fig. 2.6 shows. We use both the column number (the across number) and the row number (the number down) to specify position. For example, if we type POSITION 5,12 then this will mean column number 5 – but remember that the numbering starts at ∅, and the first column that appears on the screen is 2. The row number is 12. Again, the numbering starts at ∅, so row 12 is actually the thirteenth row counting from the top.

Figure 2.7 shows how POSITION is used along with PRINT. It

```
10 ? "}":POSITION 17,2:? "TITLE"
20 ? :?
30 POSITION 6,5:? "LOOKS A LOT BETTER
THIS WAY"
```

Fig. 2.7. Using POSITION with PRINT. Notice the 'clear screen' command. This is obtained by using ESC, then CONTROL and CLEAR. On screen, it looks like a bent arrow, but the printer reproduces it as a curly bracket.

starts by clearing the screen, using the sequence of quotemark, ESC, CONTROL and CLEAR, then quote. In the POSITION instruction, you must remember to place the comma between the two numbers that are needed. The effect of POSITION is to allow you to print items wherever you want, as Fig. 2.7 shows. You don't have to print in the order of left to right or top to bottom either, because POSITION allows you complete freedom to print wherever you want. If your choice of POSITION places a new word over an old one, then the new letters will simply replace the old ones.

If you are using POKE82,∅ to set the left margin at position ∅, the centring is carried out as follows:
Count the number of characters you want to print. Subtract from 4∅. Divide the result by 2. Subtract 1 to allow for the fact that POSITION numbering starts at ∅, not 1. As a formula, this is:

$$X = 19 - LEN(T\$)/2$$

If you are using the normal screen display, which starts at POSITION 2, then only 38 characters can be used. Use 38 in place of 4∅, and 18 in place of 19 in the formula above.

Fig. 2.8. Finding the X number for POSITION so as to centre a phrase on the screen line.

Finally, Fig. 2.8 shows how to centre a word or phrase on the screen. This assumes that you are using POSITION, and that you will have either forty or thirty-eight characters per screen line. The formula is shown in its 'BASIC' form, meaning that it's programmed just as shown. We'll look later at another centring method which does not rely on POSITION.

Chapter 3
A Bit of Variety

So far, our computing has been confined to printing numbers and words on the screen. That's one of the main aims of computing, but we have to look now at how we get to these numbers and words. This means looking at some of the actions that go on before anything is printed. One of these actions is called *assignment*.

Take a look at the program in Fig. 3.1. It starts with the 'clear

```
10 ? "}"
20 X=15
30 ? "TWO TIMES ";X;" IS ";2*X
40 X=20
50 ? "X IS NOW ";X
60 ? "AND TWO TIMES ";X;" IS ";2*X
```

Fig. 3.1. Assignment in action. The letter X has been used in place of a number.

screen' command which is programmed by using the ESC key, followed by CONTROL and CLEAR. When you type this, you see the 'bent arrow' shape on the screen. My printer shows a curly bracket instead. Type the program in, run it, and contrast what you see on the screen with what appears in the program. The first line that is printed is line 3Ø. What appears on the screen is:

2 TIMES 15 IS 3Ø

but the number 15 doesn't appear in line 3Ø! This is because of the way we have used the letter X as a kind of code for the number 15. The official name for this type of code is a *variable name*.

Line 2Ø gives X the value of 15. In computer language, we say that line 2Ø 'assigns the variable name X'. 'Assigns' means that wherever we use X, *not* enclosed by quotes, the computer will operate with the number 15. Ask the computer to PRINT X, and it will print 15. Ask it to print X+3 and it will print 18. Wherever a number can be used in

any way, X will be taken as meaning 15. Since X is a single character and 15 has two digits, that's a saving of space. It would have been an even greater saving if we had assigned X differently, perhaps as X = 2174.3256, for example. Line 3Ø then proves that X is taken to be 15, because wherever X appears, not between quotes, 15 is printed, and the 'expression' 2*X is printed as 3Ø. We're not stuck with X as representing 15 for ever, though. Line 4Ø assigns X as being 2Ø, and lines 5Ø and 6Ø prove that this change has been made.

That's why we call X a 'variable' – we can vary whatever it is we want it to represent. Until we do change it, though, X stays assigned. Even after you have run the program of Fig. 3.1, providing you haven't added new lines or deleted any part of it, you can type PRINT X, and pressing RETURN will show the value of X on the screen.

This very useful way to handle numbers in code form can use a 'name' which must start with a letter. You can add to that more letters or numbers, so that N, R2D2 and NUMBER are all 'names' that you can use for number variables, and each can be assigned to a different number. You must use upper-case (capital) letters for these names – names in lower-case or inverse video aren't accepted. You can use 'names' of as many letters as you like – within limits. The limit is 120 characters, and you would have to invent a remarkably long name to get to that limit! You are also restricted on the number of different variable names you are allowed to use. This limit is 128. Once again, you're not likely to find that this causes you sleepless nights.

Just to make it even more useful, you can use code 'names' to represent words and phrases also. The difference is that these 'names', which can also consist of more than one letter, have to have a dollar sign ($) added to the variable name. If N is a variable name for a number, then N$ (pronounced 'en-string' or 'en-dollar') is a variable name for a word or phrase. The computer treats these two, N and N$, as being entirely separate and different. The same goes for two-letter names, like NO and NO$, and also for much longer names. You can't use these variable names quite so simply, though. When you use a variable name like N, or SUM or THELOT for a number, the computer acts automatically. Your Atari 600XL codes any number so that it fits into the seven units of memory. These units of memory are called *bytes*. When you switch your Atari 600XL on, there are 13326 bytes of memory ready for you to use. There will be less if you have a disk system connected, or if you are using colour graphics (see Chapters 6 and 7). This memory is precious, because if your computer

runs out of memory, its programs can't work. To make sure that none will be wasted, the computer requires you to think how much you will need for a variable that represents letters and words. This type of variable is called a *string variable*, which is why we refer to the dollar sign as 'string'.

Figure 3.2 illustrates the use of 'string variables' as names for words and phrases. Lines 2Ø and 3Ø carry out the assignment operations,

```
10 ? "}":DIM A$(5),S$(7)
20 A$="ATARI"
30 S$="THE 16K"
40 ? S$;" ";A$
50 ? "THIS IS THE ";A$
```

Fig. 3.2. Using string variables. These are distinguished by the dollar sign, and they also have to be 'dimensioned'.

and lines 4Ø and 5Ø show how these variable names can be used. Notice that you can mix a variable name, which doesn't need quotes around it, with ordinary text, which must be surrounded by quotes. A string variable can include numbers, but a number variable can't include letters. The important thing here, though, is in line 1Ø. The computer needs to be informed of how many characters (letters or digits, punctuation marks or spaces) you might want to use in each string variable. You don't have to be exact - you might not know exactly. The point is that if you allow ten characters for a string called AS, then you must not exceed 1Ø. You can use A$="FERGUS", which is only 6 characters, but you can't use A$="CHOLD-MONDLEY", which has 12 characters. The action that appears in line 1Ø is called 'dimensioning', and it has to be done for each string name you want to use. If you forget, the computer will remind you by refusing to work with that variable! What you will see, in fact, is the screen message:

ERROR - 9 at line 2Ø

- or whatever line number contained the variable name which you had forgotten to dimension.

Figure 3.3 shows another example of the use of variable names. There wouldn't be much point in printing messages in this way if you

```
10 ? "}":DIM A$(15)
20 C=24:A$="Brown Account"
30 ? A$;" petty cash is ";C;" pounds"
```

Fig. 3.3. Using a string variable to store a message. This is very useful for messages that you want to repeat.

wanted the message once only. When, however, you are continually using a phrase in a program, this is one method of programming it so that you don't have to keep typing it! When you use a variable name to save space in this way, the use of a long variable name can make it easier for you to remember what it is that a name is supposed to represent. You might like, for example, to use variable names like ADDRESS$ or CASH$.

Stringing along

Because the name of a string variable is marked by the use of the $ sign, a variable like A$ is not confused with a number variable like A. We can, in fact, use both in the same program knowing that the computer at least will not be confused. Figure 3.4 illustrates that the difference is a bit more than skin deep, though. Lines 1Ø and 2Ø assign

```
10 A=2:B=3
20 DIM A$(2),B$(2):A$="2":B$="3"
30 ? ")"
40 ? A;" TIMES ";B;" IS ";A*B
45 ? A$;" TIMES ";B$;" IS ?"
50 ? "BUT A$*B$ IS IMPOSSIBLE_"
60 ? "THE ATARI WILL NOT ACCEPT SUCH A
 LINE"
```

Fig. 3.4. String and number variables might look alike when they are printed, but they are different!

number variables A and B, and string variables A$ and B$. When these variables are printed in lines 4Ø and 45, there isn't much difference between A and A$ or between B and B$. The important difference appears, however, in lines 5Ø and 6Ø. The computer can multiply two number variables as we've done in line 4Ø, because numbers can be multiplied, but it can't multiply string variables. The reason is simple. A string variable can be anything. We have assigned A$ as '2', but we could just as easily have assigned it as '2 ASPARAGUS AVENUE'. You can multiply 2 by 3, but you can't multiply 2 ASPARAGUS AVENUE by 3 BLACKBERRY DRIVE. The computer therefore refuses to carry out multiplication, division, addition, subtraction or any other arithmetic operation on strings. Attempting to program a forbidden operation in line 45 causes an error message to appear when you press RETURN, and the line is not entered into the memory. In this example, if you had typed line 45 so as to read:

45 PRINT A$*B$

then you would see the message

ERROR - PRINT A$*B$

appear right away, with the asterisk (multiply) sign printed in reverse video. Reverse video, remember, means that you see the asterisk printed black on a white square in place of the usual white-on-black. This should stop you from trying to enter this faulty line. If you do insist on running the program, then you will see another error message

ERROR - 17 at LINE 45

appear when you tried to RUN the program. ERROR 17 is a 'syntax error' - you have used a command wrongly. In this case, you have tried to do with a string what you can do only with a number. Later on, we'll see that there are operations that we can carry out on strings that we can't carry out on numbers, and attempts to do these operations on numbers will also cause an error message. The difference between numbers and strings is an important one. The computer stores numbers in a way that is quite different from the way it stores strings. The different methods are intended to make the use of arithmetic simple for number variables (for the computer, that is), and to make other operations simple for strings. Let's face it, it's wonderful, but it's still only a machine!

Join up, and get more!

There is one action that we can carry out on strings which is a little bit like arithmetic, though. It's called *concatenation*, but 'joining' is a lot shorter and simpler. Joining means just what it says - two strings can be joined into one, and the new one can have the variable name of one of the joined strings. As you might expect by now, there are strict rules about this sort of thing. The variable name that you are going to use for the joined up string must have been dimensioned so that it can hold all the letters of the new string. If you are going to join a string of six characters to one of seven characters, it's not much good preparing a variable name that is dimensioned to only ten characters!

Figure 3.5 illustrates this action of joining strings. This is nothing like the action of arithmetic, as you'll see if you use numbers in place of the names. The method that is used to join the strings is also rather special, and only the Atari does it this way. The instruction

```
10 DIM X$(20),A$(10),B$(10)
20 A$="ATARI"
30 B$=" 600XL"
40 X$=A$
50 X$(LEN(A$)+1)=B$
60 ? X$
```

Fig. 3.5. Concatenating or joining strings. This is not the same action as addition!

(LEN(A$)+1) means 'go to the end of A$'. When we make this equal to B$, we are placing the characters of B$ at the end of A$. We'll come back to the use of LEN later on, don't worry.

Concatenation is also a very useful way of obtaining strings which otherwise would need rather a lot of typing. Take a look at line 2Ø of Fig. 3.6. This clears the screen (using the instruction ?CHR$(125) just

```
10 DIM A$(40)
20 ? CHR$(125):A$="#":A$(37)="#":A$(2)
=A$
30 ? A$:POSITION 17,2:? "ATARI":? A$
```

Fig. 3.6. Creating a string of characters to act as a frame for a title.

for a change), and then defines string A$ as a set of characters which can be used as a 'frame' around a title. There is a piece of new programming here in the way that we produce the characters. We dimension A$ to the maximum possible number of characters we want in line 1Ø. In this example, we want thirty-eight identical characters, and the one we're using is the 'hashmark', the # sign. The title is defined in line 3Ø as ATARI. The command ?A$ then prints a concatenated string. This has needed less typing than if you had to

---

To join B$ to the end of A$:

A$(LEN(A$)+1) = B$

will carry out the merger. This causes the first character of B$ to be placed one place beyond the last character of A$. A$ must be dimensioned to the *new* length.

To fill a string with N characters, such as #:

F$="#":F$(N-1) = F$:F$(2) = F$

Once again, the string must have been dimensioned to the correct size.

Fig. 3.7. A reminder of how to create strings of identical characters, and how to join strings.

type all the characters separately. Line 2Ø that creates A$ has to be typed correctly. The first part of it assigns the character you want to string together. The next part instructs the computer about where the end of the new long string will be. This has to use a number one less than the total number you need, so we have used 38 − 1 = 37 here. The last part, A$(2) = A$ is the part which carries out the filling operation. Figure 3.7 is a reminder of what you have to do to create strings of identical characters like this. Some computers have a special instruction, STRING$ for this purpose. We don't have space in this starter book to describe how and why this action works. If you are curious, take a look at my more advanced book, *Get More From The Atari.*

Making it listen

So far, everything that has been printed on the screen by a program has had to be placed in the program before it is run. We don't have to be stuck with restrictions like this, however, because the computer allows us another way of putting information, numbers or names, into a program while it is running. A step of this type is called an INPUT and the BASIC instruction word that is used to cause this to happen is also INPUT.

Figure 3.8 illustrates this with a program that prints your name. Now I don't know your name, so I can't put it into the program

```
10 ? "}":? "WHAT IS YOUR NAME?":DIM N$
(30)
20 INPUT N$
30 ? N$;"_This is your life!"
```

Fig. 3.8. Using the INPUT instruction. The name that you type is put into the phrase in line 3Ø.

beforehand. What happens when you RUN this is that the words

WHAT IS YOUR NAME?

are printed on the first line of the screen. Another question mark is printed on the second line. Notice that you didn't put this second question mark into the program. This is something that the computer does automatically wherever there is an INPUT. The computer is now waiting for you to type something, and then press RETURN. Until the RETURN key is pressed, the program will hang up at line 2Ø, waiting for you. If you're honest, you will type your own name

and then press RETURN. You don't have to put quotes around your name, simply type it, in the form that you want to see printed. When you press RETURN, your name is assigned to the variable N$. This, like any other string variable, has to be dimensioned. Line 1Ø attends to that, using a generous thirty characters. When you have typed your name and pressed RETURN the program can then continue, so that line 3Ø then prints the famous phrase with your name at the start. Take a close look at the way in which this is programmed - the words that you want to see printed must be enclosed by quotes, and you have to specify a variable name following INPUT. If you try to ignore these rules, the Atari will stop the program at this stage with an error message.

You could, of course, have answered MICKEY MOUSE or DONALD DUCK or anything else that you pleased. The computer has no way of knowing that either of these is not your true name. There's another blow to the nutters who think that computers will take over the world! The use of INPUT isn't confined to a single name or number. We can use INPUT with two or more variables, and we can mix variable types in one INPUT line. Figure 3.9 illustrates an INPUT step in line 2Ø which uses a number variable NUMBER and a string variable NAME$. Now when the computer comes to line 2Ø, it

```
10 ? ")":DIM NAME$(20)
20 ? "Name and number, please ";:INPUT
 NAME$,NUMBER
30 ?
40 ? "The name is ";NAME$
50 ? "The number is ";NUMBER
```

Fig. 3.9. Putting in two variables in one INPUT step. Notice that you can use long names for the variables, which makes it easier for you to remember what they represent.

will print the message and the question mark, and then wait for you to enter both of these quantities, a name and then a number. You have to be particular about how you enter these quantities. You must type the name and then press RETURN. The computer will then print a question mark on the next line down. This is its way of indicating 'next one, please', and that's a signal for you to type the number and then press RETURN again. The name and number will be printed again in lines 4Ø and 5Ø. You can type items which include a comma - like Hamburger, Mike J - if you like. A lot of computers won't accept a line like this, but this is an Atari you're dealing with! There's one more thing to watch. When you use a number variable in an INPUT

step, then what you have typed when you press RETURN must be a number. If you attempt to enter a string, the computer will refuse to accept it, and you will see a message such as

ERROR - 8 in line 2Ø

appear. This will stop the program. When you use a string variable in an INPUT step, then anything - numbers, letters, punctuation marks - will be accepted.

How many beans make N?

The amount of computing that we have done so far should have persuaded you that computers aren't just about numbers. For some applications, though, the ability to handle numbers is very important. If you want to use your computer to solve scientific or engineering problems, for example, then its ability to handle numbers will be very much more important than if you bought it for games, for accounts or for word processing. It's time, then, to take a very brief look at the number abilities of the Atari 600XL. It is a brief look because we simply don't have space to explain what all the mathematical operations do. In general, if you understand what a mathematical term like SIN or TAN or EXP means, then you will have no problems about using these mathematical functions in your programs. If you don't know what these terms mean, then you can simply ignore the parts of this section that mention them.

The simplest and most fundamental number action is counting. Counting involves the ideas of *incrementing* if you are counting up and *decrementing* if you are counting down. 'Incrementing' a number means adding 1 to it; 'decrementing' means subtracting 1 from it. These actions are programmed in a rather confusing-looking way in BASIC, as Fig. 3.10 shows. Line 1Ø sets the value of variable NO as

```
10 ? "}":NO=12
20 ? "Number is ";NO:NO=NO+1
30 ? "Now it is ";NO
40 ? "Its square is ";NO^2
50 ? "Its square root is ";SQR(NO)
```

Fig. 3.10. Incrementing, using the equals sign to mean 'becomes'. Lines 4Ø and 5Ø show some other arithmetical operations.

12. This is printed in the first part of line 2Ø, but then the second part of line 2Ø 'increments NO'. This is done using the odd-looking

instruction: NO = NO + 1, meaning that the new value that is assigned to NO is 1 more than its previous value. The rest of the program proves that this action of incrementing the value of NO has been carried out.

The use of the = sign in line 2Ø to mean 'becomes' is something that you have to get accustomed to. When the same variable name is used on each side of the equality sign, this is the use that we are making of it. We could equally well have a line:

```
NO = NO - 1
```

and this would have the effect of making the new value of NO one less than the old value. NO has been decremented this time. We could also use NO = 2*NO to produce a new value of NO equal to double the old value, or NO = NO/3 to produce a new value of NO equal to the old value divided by three.

Number functions

Figure 3.10 also illustrates some number functions. A number function in this sense is an instruction which operates on a number to produce another number. Line 2Ø has changed the value of NO to 13. Line 4Ø then prints the value of NO squared, meaning NO multiplied by NO. This is programmed by typing NO^2, with the up-arrow key. The up-arrow appears on the listing as an inverted 'V'. To get the square root of the number that has been assigned to NO, we type the instruction word SQR. An alternative is NO^.5, but SQR(NO) is easier to type and remember. For other roots like the cube root you can use expressions like NO^(⅓) and so on. Notice the use of the brackets – if you leave them out, you will get an SN ERROR message when the program runs.

Figure 3.11 illustrates the various number functions that can be used, with a brief explanation of what each one does. Some of these actions will be of interest only if you want to program for scientific, technical or statistical purposes. Others, however, are useful in unexpected places, such as in graphics programs.

How precise?

One of the problems of all small computers is precision of numbers. You probably know that the fraction ⅓ cannot be expressed exactly

ABS(X)	Converts negative sign to positive.
ATN(X)	Gives the angle whose tangent has value X.
CLOG(X)	Gives the common logarithm of X.
COS(X)	Gives the cosine of angle X.
EXP(X)	Gives the value of *e* to the power X.
INT(X)	Gives the whole number part of X.
LOG(X)	Gives the natural logarithm of X.
RND(X)	Gives a random number between 0 and 1. X is not used, but a number or variable must be present.
SGN(X)	Gives the sign of X. The result is +1 if X is positive, −1 if X is negative, 0 if X is zero.
SIN(X)	Gives the sine of angle X.
SQR(X)	Gives the square root of X.

Fig. 3.11. Atari number functions, with brief notes. Don't worry if you don't know what some of these do. If you don't know, you probably don't need them!

as a decimal. How near we can get to its true value depends on the number of decimal places we are prepared to print, so that Ø.33 is closer than Ø.3, and Ø.333 is closer still. The computer converts most of the numbers it works with into the form of a fraction and a multiplier. The fraction is not a decimal fraction but a special form called a 'binary fraction', and this conversion is seldom exact. The conversion is particularly awkward for numbers like 1, 1Ø, 1ØØ and also .1, .Ø1, .ØØ1; all the powers of ten, in fact. To avoid embarrassments like printing 3 − 2 = .9999999, the computer will round numbers of this type up or down as need be before displaying them. You will sometimes find, however, that a piece of arithmetic looks spectacularly wrong, simply because it has used numbers that cannot be represented exactly inside the computer. The Atari 600XL is very much better in this respect than other computers in its price range, but it can't work miracles!

Chapter 4
Repeating the Process

A lot of the dreariest actions that we have to carry out are repeated actions. The oldest dream that people have is to be able to use a robot to carry out all the repetitive tasks that drive us out of our skulls. Every computer is therefore well equipped with instructions that will cause repetition, and the Atari is no exception. The piece of program that causes instructions to be carried out repeatedly is called a *loop*. We'll start with the simplest of these loop actions, GOTO. GOTO means exactly what you would expect it to mean - go to another line number. Normally a program is carried out by executing the instructions in ascending order of line number. In plain language, that means starting at the lowest numbered line, working through the lines in order and ending at the highest numbered line. Using GOTO can break this arrangement, so that a line or a set of lines will be carried out in the 'wrong' order, or carried out over and over again.

Figure 4.1 shows an example of a very simple repetition or 'loop', as we call it. Line 1Ø starts the program by assigning a variable N to

```
10 N=1
20 ? "__________ATARI__________"
30 ? N
40 N=N+1:GOTO 20
50 REM PRESS BREAK TO STOP
```

Fig. 4.1. A very simple loop. You can stop this by pressing the BREAK key.

have the value of 1. Line 2Ø contains a simple PRINT instruction. When line 2Ø has been carried out, the program moves on to lines 3Ø and 4Ø, which print the value of variable N, and increase it by one. The last part of line 4Ø then instructs the program to go back to line 2Ø again. This is a never-ending loop, and it will cause the screen to fill with the word ATARI, and twenty dashes, until you press the

BREAK key to 'break the loop'. Any loop that appears to be running forever can normally be stopped by pressing this key.

Now an uncontrolled loop like this is not exactly good to have, and GOTO is a method of creating loops that we would prefer not to use! We don't always have an alternative, but there is one – the FOR...NEXT loop. As the name suggests, this makes use of two new instruction words, FOR and NEXT. The instructions that are repeated are the instructions that are placed between the words FOR and NEXT. Figure 4.2 illustrates a very simple example of the FOR ...NEXT loop in action. The line which contains FOR must also

```
10 ? "}":FOR N=1 TO 10
20 POSITION 14,N:? "Atari Magic!"
30 NEXT N
```

Fig. 4.2. Using the FOR...NEXT loop for a counted number of repetitions.

include a number variable which is used for counting, and numbers which control the start of the count and its end. In the example, N is the counter variable, and its limit numbers are 1 and 10. The 'NEXT N' is in line 30, and so any instructions in lines between lines 10 and 30 will be repeated.

As it happens, what lies between these lines is simply the PRINT instruction, and the effect of the program will be to print 'Atari Magic!' ten times. At the first pass through the loop, the value of N is set to 1, and the phrase is printed. When the NEXT instruction is encountered, the computer increments the value of N, from 1 to 2 in this case. It then checks to see if this value exceeds the limit of 10 that has been set. If it doesn't, then line 20 is repeated, and this will continue until the value of N exceeds 10 – and we'll look at that point later. The effect in this example is to cause ten repetitions.

Even at this stage it's possible to see how useful this FOR...NEXT loop can be, but there's more to come. To start with, the loops that we have looked at so far count upwards, incrementing the number variable. We don't always want this, and we can add the instruction word STEP to the end of the FOR line to alter this change of variable value. We could, for example, use a line like:

```
FOR N=1 TO 9 STEP 2
```

which would cause the values of N to change in the sequence 1,3,5,7,9. We could also use an instruction like:

```
FOR N =10 TO 1 STEP -1
```

which would cause N to count downwards.

When we don't type STEP, the loop will always use increments of +1. You don't have to confine this action to single loops, either. Figure 4.3 shows an example of what we call 'nested loops', meaning that one loop is contained completely inside another one.

```
10 ? "}"
20 FOR N=10 TO 0 STEP -1
30 POSITION 10,5:? N;" Seconds and cou
nting. "
40 FOR J=1 TO 500:NEXT J
50 NEXT N:POSITION 16,10:? "BLASTOFF"
60 ? :? "Value of N is now ";N
```

Fig. 4.3. A countdown program that uses nested loops, with one loop inside another. STEP is used to make the count downwards rather than upwards.

When loops are nested in this way, we can describe the loops as inner and outer. The outer loop starts in line 2Ø, using variable N which goes from 1Ø to Ø in value. Line 3Ø is part of this outer loop, printing the value that the counter variable N has reached. Line 4Ø, however, is another loop. This must make use of a different variable name, and it must start and finish again before the end of the outer loop. We have used variable J, and we have put nothing between the FOR part and the NEXT J part to be carried out. All that this loop does, then, is to waste time, making sure that there is some measurable time between the actions in the main loop. The overall effect, then, is to show a count-down on the screen, slowly enough for you to see the changes, and printing in the same place each time. Note carefully that you have to end each loop correctly with a NEXT, and follow the NEXT with the name of the correct variable. You must also get the NEXTs in the correct order. For example, imagine that you start one loop in line 2Ø of a program, using variable X, and another in line 5Ø, using variable Y. If the second loop is to end in line 7Ø, it must use NEXT Y. Following that, in a later line, you need to end the first loop with a NEXT X. The rule is that the last loop to start must be the first one to finish – last in, first out. If you put a NEXT X in line 7Ø of our imaginary example, and NEXT Y following it, the Atari would refuse to carry out the loops. The ERROR number for this type of thing is 13.

Every now and again, when we are using loops, we find that we need to use the value of N (or whatever name we have used) after the loop has finished. It's important to know what this will be, however, and line 6Ø of Fig. 4.3 brings it home to you. This reveals that the

value of N is −1 in line 6Ø, after completing the FOR N = 1Ø TO Ø STEP −1. If you want to make use of the value of N, or whatever variable name you have selected to use, you will have to remember that it will have changed by one more step at the end of the loop.

One of the most valuable features of the FOR...NEXT loop, however, is the way in which it can be used with number variables instead of just numbers. Figure 4.4 illustrates this in a simple way.

```
10 ? "}":A=2:B=5:C=10
20 FOR N=A TO B STEP B/C
30 ? N:NEXT N
```

Fig. 4.4. A loop instruction that is formed with number variables.

The letters A, B and C are assigned as numbers in the usual way in line 1Ø, but they are then used in a FOR...NEXT loop in line 2Ø. The limits are set by A and B, and the step is obtained from an expression, B/C. The rule is that if you have anything that represents a number or can be worked out to give a number, then you can use it in a loop like this.

Decision steps

It's time to see loops being used rather than just demonstrated. A simple application is in totalling numbers. The action that we want is that we enter numbers and the computer keeps a running total, adding each number to the total of the numbers so far. From what we have done so far, it's easy to see how this could be done if we wanted to use numbers in fixed quantities, like ten numbers in a set, because this would mean starting with a FOR N = 1 TO 1Ø loop. The trouble is, how many times would you have to have just ten numbers? It would be a lot more convenient if we could stop the action simply by signalling to the computer in some way, perhaps by entering a value like Ø of 999. A value like this is called a *terminator*, something that is obviously not one of the normal entries that we would use, but just a signal. For a number-totalling program, a terminator of Ø is very convenient, because if it gets added to the total it won't make any difference.

Figure 4.5, therefore, shows an example of this type of program in action. We can't use a FOR...NEXT loop, because we don't know in advance how many times we might want to go through the loop, so we have to go back to using GOTO. This time, however, we'll keep

```
10 ? "}":POSITION 14,2:? "TOTAL FINDER
"
20 ? "The program will find the total
of":? " the numbers that you enter unt
il you":? " enter a zero."
30 TOTAL=0
40 ? "Number, please ";:INPUT NR
50 TOTAL=TOTAL+NR
60 ? "TOTAL SO FAR IS ";TOTAL
70 IF NR<>0 THEN 40
```

Fig. 4.5. A running-total program which can't use FOR...NEXT.

GOTO under closer control. We make the total variable TOTAL equal to zero in line 3Ø. Each time you type a number, then, in response to the request in line 4Ø, the number that you type is added to the total in line 5Ø, and line 6Ø prints the value of the total so far. Line 7Ø controls the loop, and the key to the control is the instruction word IF. IF is used to make a test, and the test in line 7Ø is to see if the value of NR is not equal to zero. The odd-looking sign that is made by combining the 'less than' and the 'greater than' signs, <>, is used to mean 'not equal', so the line means: 'IF NR is not equal to zero, then GOTO line 4Ø'.

The effect, then, is that if the number which you have typed in line 4Ø was not a zero, line 7Ø will send the program back to repeat line 4Ø. This will continue until you do enter a zero. When this happens, the test in line 7Ø fails (NR is zero), and the program looks for a line 8Ø. Since it can't find one, it stops.

Now this allows you much more freedom than a FOR...NEXT loop, because you are not confined to a fixed number of repetitions. The key to it is the use of IF to make a decision – and that's what we need to look at more closely now. We can make a number of types of comparisons between number variables or numbers, and these are listed in Fig. 4.6. The mathematical signs are used for convenience, and you have to remember which way round the 'greater than' and 'less than' signs have to be. It's important to note that the equals sign means 'identical to' when it is used in a test like this. If A is 5.9999999 and B is 6.ØØØØØØØ than a test such as IF A = B will fail – A is not identical to B, even though it is close enough to be equal as far as we are concerned. The important point here is that the numbers we see on the screen have been rounded, so that PRINT A in the example above might give the result 6. The test, however, is made on the numbers which have not been rounded.

Sign	Meaning
=	Exact equality.
>	Left-hand quantity greater than right-hand quantity.
<	Left-hand quantity less than right-hand quantity.
The signs can be combined as follows:	
<>	Quantities not equal.
>=	Left-hand quantity greater than or equal to right-hand quantity.
<=	Left-hand quantity less than or equal to right-hand quantity.

Fig. 4.6. The mathematical signs that are used for comparing numbers and number variables.

Figure 4.7 shows another test – this time on string variables. The instructions are in line 2Ø; you are asked to press the Y or N key. Line 3Ø gets your answer; you have to type Y or N and then press RETURN. The key that you have pressed has its value assigned to

```
10 ? "}":DIM A$(1)
20 ? "Please press Y or N key and then
 ":? "press RETURN."
30 INPUT A$
40 IF A$="Y" THEN ? "That's YES":GOTO
70
50 IF A$="N" THEN ? "That's NO":GOTO 7
0
60 ? "You cheated _ try again":GOTO 20
70 ? :? "That's it!"
```

Fig. 4.7. Testing string variables, in this example to find whether a reply is Y or N.

A$, so that A$ should be Y or N. Lines 4Ø and 5Ø then analyse this result. If both tests fail, though, the program will move from line 5Ø to line 6Ø. Your answer was not exactly Y or N, so that you are asked to try again, and the GOTO 2Ø at the end of line 6Ø causes the program to repeat from line 2Ø. This line 6Ø constitutes a *mugtrap*, a way of trapping mistakes. Very often, when you have a choice of answers, you want to be sure that only certain replies are permitted. A mugtrap is a section of program that is intended to deal with an incorrect entry. A good mugtrap should show the user the error of his or her ways, and indicate what answer or answers might be more acceptable. This is very often important, because an incorrect entry in some types of program could cause the program to stop with an error

message showing. For the more skilled programmer (as you will be by the time you finish this book), this is just a minor annoyance, but for the inexperienced user it can cause a minor panic. A good program doesn't allow any entries that would cause the program to stop. Mugtraps are our method of ensuring this.

The test in this example is for identity. Only if A$ is absolutely identical to Y will the phrase 'That's YES' be printed. If you typed a space ahead of Y, or a space following, or typed y in place of Y, then A$ will not be identical, and the test fails. Failing means that A$ is not absolutely identical to Y and everything that follows THEN in that line will be ignored. It's up to you to form these tests so that they behave in the way that you want.

Just to emphasise the sort of power that these simple instructions give you, Fig. 4.8 illustrates a very elementary number-guessing game. Line 10 starts by clearing the screen, and printing a title. The X

```
10 ? "}":? :? "Guess the number":SCORE
=0
20 ? :? "If you get near, I'll tell yo
u"
30 X=1+INT(RND(0)*10)
40 INPUT N
50 IF N=X THEN ? "SPOT ON!":SCORE=SCOR
E+5:GOTO 80
60 IF ABS(N-X)<3 THEN ? "Close_ it was
 ";X:SCORE=SCORE+2:GOTO 80
70 ? "Try again _ ":GOTO 40
80 ? "Your score is ";SCORE:? "Try ano
ther one (press BREAK to stop)":GOTO 3
0
```

Fig. 4.8. A simple number-guessing game which uses number comparisons.

= 1+INT(RND(0)*10) step then causes variable X to take a value that lies between 1 and 10. We can't predict what this value will be, because RND means 'select at random'. What RND actually does is to generate a number whose value is always a fraction, something between almost zero and almost 1. Now a randomly-picked fraction isn't of much interest to us, but we can convert it into a randomly picked whole number! What we do is to multiply this random fraction by ten. This gives a number that can be anything from less than 1 to just under ten. This will still have a fractional part, though. If we take the INT of this number (INT means integer-part, the whole number part), then the number will be anything from 0 to 9. Add 1 to this, and we have a range of 1 to 10 – picked at random. Just to

illustrate it, suppose that in two tries, RND generated first of all Ø.ØØ1236 and then .998764. Multiply each of these by ten, and you get Ø.Ø1236 and 9.98764. Take the whole number part, and you get Ø and 9. Add 1 and you get 1 and 1Ø. Try it for yourself with any randomly-picked fraction. Remember that when you have brackets in a formula like this, whatever is inside brackets is worked out first. The Ø that follows RND is what is called a 'dummy variable'. It doesn't do anything, but the computer needs to have a number following a number function like RND.

In line 2Ø, then, the instructions ask you to guess the size of the number, with the difference that you don't have to find it exactly. You enter your number at line 4Ø, and the tests are made in lines 5Ø and 6Ø. If the number that you picked is identical to the random number, then you get the SPOT ON! message in line 5Ø, and the program adds five points to your score and moves to line 8Ø. The less obvious test is in line 6Ø. The expression N − X is the difference between your guess, N, and the number X. If your guess is larger than the number, then N − X is a positive number. If your guess is less than X, then N − X is a negative number. The effect of ABS, however, is to make any number positive, so that if X were 5 and you guessed 6 or 4, then ABS (N − X) would come to 1 either way round. If you get a difference of 1 or 2 (less than 3), the message in line 6Ø is printed. The program then presents you with two points, and shifts to line 8Ø. If you don't get anywhere near, everything following THEN in line 60 is ignored, and line 7Ø is carried out. This takes you back to line 4Ø, and the program repeats the INPUT step. When you have scored, however, the program goes back to line 3Ø to generate a new random number. It's very simple, but quite effective. Because the loops are endless, you have to press BREAK to stop the program. Could you improve on that?

Reading the data

There's yet another way of getting data into a program while it is running. This one involves reading items from a list, and it uses two instruction words READ and DATA. The word READ causes the program to select an item from the list. The list is marked by starting each line of the list with the word DATA. The items of the list can be separated by commas. Each time an item is read from such a list, a 'pointer' is altered so that the next time an item is needed, it will be the

next item on the list. The READ...DATA instructions really come into their own when you have a long list of items that are read by repeating a READ step.

The program in Fig. 4.9 illustrates how we can use these instructions. Line 2Ø starts a loop which will select ten items. In line

```
10 ? "}":? "Number",,"Root"
20 FOR N=1 TO 10
30 READ J
40 ? J,,SQR(J)
50 NEXT N
60 DATA 23,14,11,43,224,10715,56,72,84
,59
```

Fig. 4.9. Using READ and DATA. These are useful when you want to keep numbers and words ready for use.

3Ø, the instruction READ J means that a number will be selected from a list, and assigned to the variable name J. The list of numbers is in line 6Ø. It is marked by the word DATA, and the computer will completely ignore this line until it comes to the READ part of the instruction. As each number is read in turn, the number and its square root is printed in line 4Ø, and the next number is read. We have to be careful to match the number of items that follow DATA with the number of times we use the loop. If we try to READ eleven items, and have provided only ten, then an error message will put a stop to the program for us. The error number is 6, meaning 'out of DATA'. We also have to match the data items with the variable names that we use for them. We can read a number item and assign it to a string variable name, but we can't read a string item and assign it to a number variable. We can, however, use several lines of DATA, and the computer will read the items in order, starting at the lowest numbered line.

The benefits of READ...DATA are not confined to numbers alone. Figure 4.10 illustrates a use of READ and DATA with strings. The aim of the program is to find out how much your shopping costs! It's based on the totalling program that we looked at earlier, with the difference that a READ takes place in line 2Ø. This reads an item from the list in line 1ØØ and tests to find if the item is called 'END'. If the item is not 'END' then line 3Ø prints its name, and calls for the number and price to be entered in line 4Ø. The program loops round, calculating the total cost in line 5Ø each time, until the 'END' item is read. When this happens, line 7Ø gives the total cost, and the program

```
10 ? "}":TOTAL=0:DIM A$(12)
20 READ A$:IF A$="END" THEN 70
30 ? "How many ";A$;" do you want?":?
"_ and at what price?"
40 INPUT NUMBER,COST
50 TOTAL=TOTAL+NUMBER*COST
60 GOTO 20
70 ? "Total cost will be ";TOTAL
100 DATA Apples,Bananas,Pears,Apricots
, Peaches,Peanuts,Oranges, Lemons,Grap
efruit,END
```

Fig. 4.10. Using READ and DATA with strings. You must use a string variable, but the words don't need quotes around them.

ends. There's one more twist to the use of the READ and DATA instructions, in the form of RESTORE. When you have read all of the DATA items, any attempt to READ again will cause the program to stop with an error message (Error 6). If you type RESTORE in a program line, this will return the DATA list to the first item again, so you can repeat all the reading actions again. You can't, however, return to a selected item, only to the first one. The main use of READ, DATA and RESTORE is for putting in values that a program has to use over and over again. For new data, you would normally use INPUT.

Single key reply

So far, we have been putting in Y or N replies with the use of INPUT, which means pressing the key and then pressing RETURN. This has the advantage of giving you time for second thoughts, because you can delete what you have typed and type a new letter before you press RETURN. For snappier replies, however, there is an alternative in the form of GET. GET is an instruction that carries out a check of the keyboard to find if a key is pressed. If no key has been pressed, the computer hangs up, repeating its check over the keys, until a key is pressed. When we use GET, we have to prepare for it. The preparation involves signalling to the computer that we will be looking for signals from the keyboard. This is done by using, early in the program, the instruction OPEN #1,4,Ø,"K:". This is called 'opening a channel', and it must be typed exactly as shown. The 'K' is the part that indicates that we want it to search for something coming from the keyboard.

Figure 4.11 shows GET in use. The line that uses GET #1,X will

```
10 ? "}":OPEN #1,4,0,"K:"
20 ? :? "Press any key"
30 GET #1,X
40 ? :? X;" is the number."
50 ? :? "It's the ";CHR$(X);" key."
```

Fig. 4.11. Using GET to find when a key has been pressed. Some keys will cause the program to operate, but will not print anything on the screen.

cause the computer to hang up, waiting for a key to be pressed. When you press a key, the variable X is assigned a number. What number? A number that is a code for that key. Any code could be used, but the one that is used is the code called ASCII. The letters mean American Standard Code for Information Interchange. This ASCII code is one that is almost universally used by computers, printers, and all sorts of devices that fit on to computers. Figure 4.12 shows a list of these ASCII codes. It's something that we'll make a lot of use of later, too.

Subroutines and menus

A subroutine is a section of program which can be inserted anywhere that you like in a longer program. A subroutine is called into action by typing the instruction word GOSUB, followed by the line number in which the subroutine starts. When your program comes to this instruction, it will jump to the line number that follows GOSUB, just as if you had used GOTO. Unlike GOTO, however, GOSUB offers an automatic return. The word RETURN is used at the end of the subroutine lines, and it will cause the program to return to the point that immediately follows the GOSUB. Figure 4.13 illustrates this. When the program runs, line 1Ø prints a phrase, with the semicolon used to prevent a new line from being selected. The GOSUB1ØØØ in line 2Ø then causes the word 'yellow' to be printed, but the RETURN in line 1Ø1Ø will send the program back to line 3Ø, the instruction that immediately follows the first GOSUB1ØØØ. This action will also occur even when the GOSUB is part of a multistatement line, as lines 4Ø and 5Ø demonstrate. The GOSUB1ØØØ will cause the word 'yellow' to be printed, but the return is to the PRINT instruction that follows GOSUB1ØØØ in line 5Ø, it doesn't jump to line 6Ø.

Now to see one of the most useful applications of subroutines, we have to hark back to choices. A choice of two items, such as in Fig. 4.7 isn't exactly generous. We can extend the choice by a program routine that is called a *menu*. A menu is a list of choices, usually of

Code	Character	Code	Character	Code	Character
32	Space	64	@	96	
33	!	65	A	97	a
34	"	66	B	98	b
35	#	67	C	99	c
36	$	68	D	100	d
37	%	69	E	101	e
38	&	70	F	102	f
39	'	71	G	103	g
40	(	72	H	104	h
41	)	73	I	105	i
42	*	74	J	106	j
43	+	75	K	107	k
44	,	76	L	108	l
45	-	77	M	109	m
46	.	78	N	110	n
47	/	79	O	111	o
48	0	80	P	112	p
49	1	81	Q	113	q
50	2	82	R	114	r
51	3	83	S	115	s
52	4	84	T	116	t
53	5	85	U	117	u
54	6	86	V	118	v
55	7	87	W	119	w
56	8	88	X	120	x
57	9	89	Y	121	y
58	:	90	Z	122	z
59	;	91	[	123	
60	<	92	\	124	|
61	=	93	]	125	
62	>	94	^	126	
63	?	95	_	127	

Fig. 4.12. The ASCII codes, which are the numbers that the GET instruction uses.

```
10 ? "}":? "This is a ";
20 GOSUB 1000
30 ? "subroutine":? :?
40 ? "Red light and green light make "
;:GOSUB 1000:? " light."
50 ? "Wasps have ";:GOSUB 1000:? "and
black stripes."
60 END
1000 ? " yellow ";
1010 RETURN
```

Fig. 4.13. Using a subroutine – this is the key to more advanced programming.

program actions. By picking one of these choices, we can cause a section of the program to be run. One way of making the choice is by numbering the menu items, and typing the number of the one that you want to use. We could use a set of lines such as

IF K = 1 THEN 1ØØØ
IF K = 2 THEN 2ØØØ

and so on. Figure 4.14 shows an improvement on this one, making use of subroutines. It's a feature of Atari that we can use the

```
10 ? "}":OPEN #1,4,0,"K:"
20 POSITION 14,1:? "PROGRAMS MENU"
30 ? :? "1. Enter program name."
40 ? "2. Enter amount of RAM used."
50 ? "3. List all programs."
60 ? "4. List programs below specified
 size."
70 ? "5. END."
80 ? :? "Please select by number. ":?
"Do NOT use RETURN key."
90 GET #1,K
100 IF K<49 OR K>53 THEN ? "Faulty sel
ection_1 to 5 only":?`"Please try agai
n.":GOTO 90
110 ON K-48 GOSUB 1000,2000,3000,4000,
5000
130 ? "That's the END"
140 END
1000 ? "Name section":RETURN
2000 ? "RAM section":RETURN
3000 ? "List-all section":RETURN
4000 ? "Select section":RETURN
5000 RETURN
```

Fig. 4.14. A menu choice that makes use of subroutines.

instruction ON N GOSUB. We don't, of course, have to use N, any variable can be used here. The menu prints the choices and allocates

each one to a number. The GET routine in line 9Ø waits for you to press a key – but the value of K that it obtains is an ASCII code. This ASCII code value has to be converted back into a number value in line 11Ø by using K – 48. If you hark back to the ASCII codes, you'll see that 48 is the code for zero. If you pressed the '1' key, then, it would give K the value of 49, and 49 – 48 is 1, just what we want! Line 11Ø tests the value of K before we go to the bother of converting it, to make sure that it is of the right size. We know that we can't accept values less than 49 ('1' pressed) or greater than 53 ('5' pressed). I haven't written out the GOSUB lines in full, because that would be just too much typing for the sake of a demonstration. The PRINT lines, however, prove that the subroutine action has been carried out.

The key to the action is ON K GOSUB. It is followed by a list of line numbers. If K = 1, then the first of these line numbers is used. If K = 2, then the second line number is used, and so on. Providing that your list of line numbers following ON K GOSUB is in the same order as your 'choice numbers' of 1,2,3, and so on, the action is automatic. Though I have illustrated with line numbers 1ØØØ to 5ØØØ, you could have had a line such as:

```
ON K – 48 GOSUB136Ø,1122,5Ø1Ø,2Ø,42
```

as long as these were the correct numbers of the subroutines.

Note: Subroutine lines should be placed *after* the END of the program. If you don't do this, they can be run whenever the program reaches those particular line numbers. The program will then stop with an error message when it reaches the first RETURN.

Chapter 5
Stringing along with Atari

In Chapter 3, we took a fairly brief look at number functions. If numbers turn you on, that's fine, but string functions are in many ways more interesting. What makes them that way is that the really eye-catching and fascinating actions that the computer can carry out are so often done using string functions. What's a string function, then? As far as we are concerned, a string function is any action that we can carry out with strings. That definition doesn't exactly help you, I know, so let's look at any example. Figure 5.1 shows a program that prints ATARI MAGIC as a title. What makes it more eye-

```
10 ? "}":DIM T$(15),A$(20)
20 T$="ATARI MAGIC"
30 GOSUB 1000
40 ?
50 T$="#":T$(10)="#":T$(2)=T$
60 GOSUB 1000
70 END
1000 X=INT(20-LEN(T$)/2)
1010 A$=" ":A$(X-3)=" ":A$(2)=A$
1020 ? A$;T$:RETURN
```

Fig. 5.1. Introducing LEN, a member of the string function family, to print titles centred. The value of X gives the number of spaces that we need in front of the word, and this number can then be packed into a string A$.

catching is the fact that the words are printed with an underlining of # shapes. This uses the same joining up instruction as we met first in Chapter 3.

Now what this program does is to make use of a subroutine to print ATARI MAGIC and its underlining of hashmarks centred along each line. The centring is carried out in line 1Ø2Ø, by printing a set of spaces ahead of the variable T$. The set of spaces is the string A$. Line 1ØØØ finds how many spaces are needed, and this is the line that

introduces the first string function of this chapter. LEN is used to find how many characters (letters, digits, punctuation marks or spaces) exist in a string variable. LEN(T$) will come up with the number of characters in the variable T$, whatever T$ may be. What we're doing in the subroutine, then, is to find how many characters are in the title. We take half of this number, and subtract it from 2Ø. The resulting number is then used in line 1Ø1Ø to pack a string, A$, with spaces - note that we use X - 3, not X. Notice also that if we want anything printed centred by this subroutine, we have to give it the variable name of T$. This action is called 'passing a variable' to the subroutine, and it's somthing that we have to keep a careful eye on when we use subroutines. You can't expect a subroutine that is written to print T$ centred to have any effect on a string called Y$, for example.

A slice in time

The next group of string operations that we're going to look at are called slicing operations. The result of slicing a string is another string, a piece copied from the longer string. String slicing is a way of finding what letters or other characters are present at different places in a string. All of that might not sound terribly interesting, so take a look at Fig. 5.2. The string A$ is assigned in line 2Ø, and sliced in lines 3Ø, 4Ø and 5Ø. What's printed in line 6Ø is the word ATARI. Now how did this happen? We need to examine lines 3Ø to 5Ø in detail to see how the numbers in the brackets have achieved the effect that you see.

```
10 DIM A$(15),B$(10)
20 A$="AT WAR IN VAIN"
30 B$=A$(1,2)
40 B$(3)=A$(5,6)
50 B$(5)=A$(8,8)
60 ? B$
```

Fig. 5.2. Using the Atari string slicing actions. The numbers within the brackets are for the start and end character, respectively.

We have to start by imagining each character in a string numbered, starting with 1 at the left-hand side. We then count each letter, digit, space or punctuation mark that follows it. Now to copy a slice out of a string, we have to specify where we start to cut, and where we end. It's done using these position numbers. Suppose, for example, that we

have the word GALLOP (see Fig. 5.3) and it's assigned to A$. Did you dimension A$ to take it? Good, then we can slice it. Suppose we type PRINT A$(1,3). Position 1 is where the 'G' is, and position 3 is where the first 'L' is. What we get, then, is 'GAL', letters 1 to 3. Take another example, A$(2,4). Letter 2 is 'A', letter 4 is the second 'L', so A$(2,4) is 'ALL'. Now, what's A$(2,5) repeated twice?

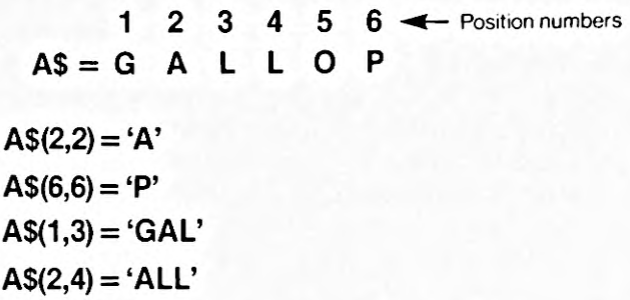


Fig. 5.3. Illustrating how slicing works.

ASC and CHR$

Some of the most useful string operations make use of the ASCII code numbers that we have already met. We can find out the code for any letter by using the function ASC, which is followed, within brackets, by a string character. The result of ASC is a number, the ASCII code number for that character. If you use ASC(A$), where A$ is a collection of characters, then you'll get the code for the first character only, because the action of ASC includes rejecting more than one character. Figure 5.4 shows ASC in action. String variable

```
10 ? "}":DIM A$(20)
20 A$="ATARI 600XL"
30 FOR J=1 TO LEN(A$)
40 ? ASC(A$(J,J));" ";
50 NEXT J
```

Fig. 5.4. Using ASC to find the ASCII code for letters.

A$ is assigned in lines 20 and in line 30 a loop starts which will run through all the letters in A$. The letters are picked out one by one, using the slicing action to select a different letter each time. This is done by using the loop counting variable J as the number for slicing, using A$(J,J) to slice just one letter each time. The ASCII code for

each letter is then found with ASC. The space between quotes, along with the semicolons in line 4Ø makes sure that the codes are all printed on one line with a space between the numbers. Simple, really.

ASC has an opposite function, CHR$. What follows CHR$, within brackets, has to be a code number, and the result of the action is the character whose code number is given. The instruction PRINT CHR$(65), for example, will cause the letter A to appear on the screen, because 65 is the ASCII code for the letter A. We can use this for hiding messages. Every now and again, it's useful to be able to hide a message in a program so that it's not obvious to anyone who reads the listing. Using ASCII codes is not a particularly good way of hiding a message from a skilled programmer, but for non-skilled users it's good enough. Figure 5.5 illustrates this use. Line 5Ø contains

```
10 ? "}":OPEN #1,4,0,"K:"
20 ? "What's the word for computer?"
30 ?
40 ? "Press any letter key to reveal."
50 GET #1,X
60 ?
70 FOR J=1 TO 5:READ N
80 ? CHR$(N);
90 NEXT J
100 END
110 DATA 65,84,65,82,73
```

Fig. 5.5. Using ASCII codes to carry a coded message, and then using CHR$ to obtain the character that corresponds to a code number.

a GET loop to make the program wait for you. When you press a key, the loop that starts in line 7Ø prints 5 characters on the screen. Each of these is read as an ASCII code from a list, using a READ instruction in the loop. The PRINT CHR$(N) in line 8Ø then converts the ASCII codes into characters and prints the characters, using a semicolon to keep the printing in a line. Try it! If you wanted to conceal the letters more thoroughly, you could use quantities like one quarter of each code number, or 5 times each code less 2Ø, or anything else you like. These changed codes could be stored in the list, and the conversion back to ASCII codes made in the program. This will deter all but really persistent decoders!

The big advantage of CHR$, of course, is that we can use it with numbers that are not ASCII codes. We have already used CHR$(125) as a way of clearing the screen in a program. There are several more codes of this type, which are 'non-printing' codes. In addition, ASCII numbers above the value of 128 are used for graphics characters,

shapes rather than letters or digits. We'll come on to these in the next chapter, so be patient!

STR$ and VAL

We saw earlier that string variables and number variables are completely different, even if a string consists of a number. For such examples, numbers that are stored in string form, we have a special pair of commands, STR$ and VAL. Taking the second one first, VAL converts a number from string form into number form. If you have, for example, a number stored in string form as variable NR$, then you can't carry out arithmetic with it. If you use VAL (NR$), however, you can carry out arithmetic. The opposite of this is STR$. A number variable like A can be converted into string form by using, for example, NR$ = STR$(A). Figure 5.6 illustrates STR$ and VAL in action.

```
10 ? "}":DIM N$(5),V$(5)
20 N$="22.5":V=2.5
30 ? N$;" times ";V;" is ";V*VAL(N$)
40 ?
50 V$=STR$(V)
60 ? "There are ";LEN(V$);" characters
 in ";V
70 ?
80 ? N$;" added to ";V$;" gives ";VAL(
N$)+VAL(V$)
```

Fig. 5.6. Using STR$ and VAL for converting between string and number form.

Order! Order!

We saw earlier, in Fig.4.8, how numbers can be compared, and we have also looked at the idea of testing strings for equality. We can also compare strings, using the ASCII codes as the basis for comparison. Two letters are identical if they have identical ASCII codes, so it's not difficult to see what the identity sign, =, means when we apply it to strings. If two long strings are identical, then they must contain the same letters in the same order. It's not so easy to see how we use the > and < signs until we think of ASCII codes. The ASCII code for A is 65, and the code for B is 66. In this sense, A is 'less than' B, because it has a smaller ASCII code. If we want to place letters into alphabetical

order, then, we simply arrange them in order of ascending ASCII codes.

This process can be taken one stage further, though, to comparing complete words, character by character. Figure 5.7 illustrates this use of comparison using the = and > symbols. Line 2Ø assigns a nonsense

```
10 ? "}":DIM A$(20),B$(20),C$(20)
20 A$="QWERTY"
30 ? :? "Type a word ";:INPUT B$
40 IF B$=A$ THEN ? "Same as mine!":END

50 IF A$>B$ THEN C$=A$:A$=B$:B$=C$
60 ? "Order is ";A$;" then ";B$
70 END
```

Fig. 5.7. Comparing words to decide on their alphabetical order.

word – it's just the first six letters on the top row of letter keys. Line 3Ø then asks you to type a word. The comparisons are then carried out in lines 4Ø and 5Ø. If the word that you have typed, which is assigned to B$ is identical to QWERTY, then the message in line 4Ø is printed, and the program ends. If QWERTY would come later in an index than your word then line 5Ø is carried out. If, for example, you typed POLYGON, then since P comes before Q in the alphabet and has an ASCII code that is lower than the code for Q, the word A$, which is QWERTY, scores higher than B$, and line 5Ø swaps them round. This is done by assigning a new string, C$ to A$ (so that C$ = "QWERTY"), then assigning A$ to B$ (so A$ = "POLYGON"), then B$ to C$ (so that B$ = "QWERTY"). If the word that you typed 'comes later' than QWERTY, meaning that its ASCII codes are higher, then A$ is not 'greater than' B$. Suppose, for example, that you type 'TAPE' as your word. This comes 'later than' QWERTY and the second part of line 5Ø will be ignored. Line 6Ø will then print the words in the order A$ and then B$, which will be the correct alphabetical order. Note the important point, though, that words like QWERTZ and QWERTX will be put correctly into order – it's not just the first letter that counts. In addition, lower-case (small) letters have ASCII code numbers that are 32 greater than their upper-case versions. The ASCII code for A is 65, but the ASCII code for a is 65 + 32 = 97. This means that BOOK will always appear ahead of book when you use the computer to arrange words.

Numbers in array

The variable names that we have used so far are useful, but there's a limit to their usefulness. Figure 5.8 illustrates this. Lines 1Ø to 4Ø generate an (imaginary) set of examination marks. This is done using

```
10 ? "}":DIM A(10)
20 FOR N=1 TO 10
30 A(N)=10+INT(90*RND(0))
40 NEXT N
50 ?
60 ? "   MARKS LIST"
70 ? "   ##########"
80 FOR N=1 TO 10
90 ? "Item ";N;" received ";A(N);" mar
ks."
100 NEXT N
```

Fig. 5.8. An array of subscripted number variables. It's simpler than the name suggests!

random numbers simply to avoid the hard work of entering the real thing. No, it's not the way that the examiners usually arrive at their marks. The variable A(N) in line 3Ø is something new, though. It's called a subscripted variable, and the 'subscript' is the number that is represented by N. The name that we use has nothing to do with computing, it's a name that was used long before computers were around. How often do you make a list with the items numbered, 1,2,3 ... and so on? These numbers 1,2,3 are a form of subscript number, put there simply so that you can identify different items. Similarly, by using variable names A(1), A(2), A(3) and so on, we can identify different items that have the common variable name of A. The whole group is called an *array*. A member of this group like A(2) has its name pronounced as 'A-of-two'.

The usefulness of this method is that it allows us to use one single variable name for the complete list, picking out items simply by their identity numbers. Since the number can be a number variable or an expression, this allows us to work with or pick out any item from the list. Figure 5.8 shows the list being constructed from the FOR...NEXT loop in lines 2Ø to 4Ø. Each item is obtained by finding a random number between 1Ø and 99, and is then assigned to A(N). Ten of these 'marks' are assigned in this way, and then lines 6Ø to 1ØØ print the list. It makes for much neater programming than you would have to use if you needed a separate variable name for each number.

You can't just rush into the subscripted variable business, though.

The computer has to keep track of all these different values, and this needs a certain amount of organisation. This means instructing the computer to prepare space. This is done by using the instruction word DIM once again. DIM, remember, means 'dimension', and the instruction consists of naming each variable that you will use for arrays, and following the name with the maximum number, within brackets, that you expect to use. You aren't forced to use this number, but you must not exceed it. If you attempt to use a number that is higher than the one you have put into the DIM instruction, then the computer will stop with an error message ERROR - 9. You will have to change the DIM instruction and start again - which will be tough luck if you were typing in a list of 1ØØ names ...! You have to dimension for every array that you use. DIM must be carried out as early in the program as possible, and you must not attempt to carry out another DIM on the same variable in the same program.

Some computers allow 'string arrays', using the same arrangement for strings as we use for number arrays. The Atari does not use string arrays. Instead, you can create long strings that consist of several strings joined together, and you can do with such a long string almost anything that you could do with a string array. There's more than one way of doing this. In this book we'll keep to a comparatively simple method. A more advanced way is illustrated in my other book, *Get More From The Atari.*

To start with, we make every string of the same length. They don't have to start that way. If you are making a list of names, for example, you might have names as short as SAM and as long as ROSALIND. No matter, we'll make them all ten characters long. How do we do it? We simply add spaces after the last letter of each name until the name is ten characters long! We have already seen this in action, and Fig. 5.9 shows this packing action in progress. Each time you enter a name, it is padded to length, and it's printed, along with a note of its length. Yes, they all end up as ten characters long!

Now this padding action makes it easy to create a long string. Though you aren't allowed to type a string longer than 120 characters, there's nothing to stop you from making a string as long as you like by joining up shorter strings. Nothing, that is, provided that you have dimensioned the variable name that you are going to use. We can, for example, dimension a string variable name like X$ to take 1000 characters. Now suppose we have an input step that asks

```
10 DIM A$(10),B$(10),C$(10),D$(10),X$(
15)
20 GOSUB 1000:A$=X$
30 GOSUB 1000:B$=X$
40 GOSUB 1000:C$=X$
50 GOSUB 1000:D$=X$
60 ? ")":DIM Y$(12):Y$="length is "
70 ? :?
80 ? A$;" ";Y$;LEN(A$)
90 ? B$;" ";Y$;LEN(B$)
100 ? C$;" ";Y$;LEN(C$)
110 ? D$;" ";Y$;LEN(D$)
120 END
1000 ? :? "Type a name- no more than 1
2 letters."
1010 INPUT X$:L=LEN(X$)
1020 IF L<10 THEN X$(L+1,10)="
     "
1030 X$=X$(1,10):RETURN
```

Fig. 5.9. How to create a long string of 'packed' names.

you to type a name, and that this name is padded out to ten letters, and assigned to the variable A$. By having the step X$(1) = A$, we make the letters of A$ take up the first ten places of the string X$. Now if we take another A$, another padded name, we can use X$(11) = A$ to place the letters of this new string into X$, taking positions 11 to 2Ø. Yes, we can use a loop for all this. A loop can be used to adjust the numbers, too. If we us Y as the number, then it will have to start at 1. If we add ten to it each time we have another string to add, though, it will automatically go to 11,21,32, and so on. These are the right positions in X$ for starting another insertion. The result is shown in diagram form in Fig.5.10.

Figure 5.11 extends this another step further. This is a program that allows you to create a string of ten names. You are invited to type

Position:	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23
Character:	A	L	I	C	E	□	□	□	□	□	P	E	T	E	R	□	□	□	□	□	R	I	C
Position:	24	25	26	27	28	29	30	31	32	33	34	35	36	37	38	39	40	41	42	43	44		
Character:	H	A	R	D	□	□	□	S	U	E	□	□	□	□	□	□	□	J	U	L	I		
Position:	45	46	47	48	49	50	51	52	53	54	55	56	57	58									
Character:	A	N	□	□	□	□	T	O	M	□	□	□	□	□									

etc.

Note: □ means space

Fig. 5.10. Packing strings with blanks so that they are all of the same length.

```
10 DIM A$(15),X$(100):X$=""
20 ? "}":? :? :FOR N=1 TO 10
30 GOSUB 1000:X$(LEN(X$)+1)=A$
40 NEXT N:? :? X$
50 FOR J=1 TO 5000:NEXT J
100 FOR N=0 TO 8 STEP 2:FOR S=0 TO 1
110 Z=10*(N+S)+1
120 ? X$(Z,Z+9),
130 NEXT S:? :NEXT N
140 END
1000 ? :? "Please type a name -up to 1
2 letters."
1010 INPUT A$:L=LEN(A$)
1020 IF L<10 THEN A$(L+1,10)="
  "
1030 A$=A$(1,10):RETURN
```

Fig. 5.11. A program that creates a long string of names, and also 'unpacks' it!

a name for each of ten items. After you have pressed RETURN, the name that you have typed is assigned to A$, packed to ten characters, and added to the string X$. When the list is complete, there is a pause so that you can get your breath back. The pause is programmed by using a FOR ... NEXT loop by itself, with nothing done between the FOR and the NEXT. The screen is then cleared. The list is then printed neatly, using line 12Ø. This illustrates another nested loop. The loop that starts in line 1ØØ uses the variable 'S'. When S is zero, and the value of N is also Ø, for example, then the value of Z in line 11Ø is 1Ø*Ø+1, which is just 1. What is printed in line 12Ø, then, is X$(1,1Ø), which is the first name. The comma then forces printing to the centre of the screen, and the NEXT S takes effect. This causes the value of N + S to become 1, so that Z is 11, and X$(11,2Ø) is printed. That's the end of the S loop this time round, so we go back for another value of N, taking a new line. The new line is automatically selected because of the use of the ? (PRINT) between the NEXT S and the NEXT N. The important point about this program is that it demonstrates that it's not just numbers that we can keep in the form of an array. The long string that we have used here can have the same name as a number variable or a number array. This will not confuse the computer, because it treats these different types of variables very differently. For example, you can have a number variable N, a string variable N$, and a number array N(X) existing together. The Atari will keep track of these different types of variables and will never confuse them – though you might!

As a matter of record

Entering items into arrays is hard work, particularly if you are a one-finger typist. It's certainly an activity that you don't want to repeat if you can possibly avoid it. Atari allows you to avoid it by saving (which means recording) any type of variables on cassette. The much superior method is to save programs and data on magnetic disks, and these are available for the Atari right now. For the moment, though we'll stick to the use of cassettes. When you record your programs on cassettes, the program variables are recorded as well. Suppose you have a program in which variables X, Y and Z$ are used. When you RUN this program, these variable names will be assigned with values. If you now record the program, using CSAVE, then the variable values will be recorded along with the program. This isn't obvious if you type RUN after loading the program, because RUN always erases any variable values. If, immediately after a CLOAD operation, you type something like ?X:?Y:?Z$, and then press RETURN, you will see the values printed. If you want to resume a program with these values, you can type GOTO and follow this with a line number. The program will start at this line, using the variable values that were recorded. It's a useful feature – apart from anything else, it allows you to record a game while you are halfway through, and carry on with it next time! You can't, however, do this quite so easily if you use disks, because program variables are not recorded on the disk.

The Atari also allows you to record and replay the values of variables quite separately from a program. This way you can have variable values created by one program, and used by another. If you have a program that creates a list of names, addresses, and birthdays of your friends, for example, you can use the data (the names and addresses) in lots of ways. You can have a program that makes party invitations, one that prints Christmas cards, one that reminds you of birthdays, and so on. All of these different programs can use the information that was created by the first program.

The program in Fig. 5.12 illustrates how variables – two strings and a number in this example – can be saved on tape. The steps up to line 4Ø should be familiar territory to you by now, so we'll concentrate on 4Ø onwards. Line 4Ø delivers a message to tell you what's happening. This is essential, because you can't make a recording until you have a cassette in the recorder – with no reminder like this, you could easily forget. Line 5Ø is the start of the data-saving lines. The keyword is OPEN, and it has to be followed by four items. The first of these is #1,

```
10 ? ")":DIM A$(5),B$(50)
20 A$="ATARI":? "Please type your full
 name."
30 INPUT B$:C=3.142
40 ? :? "Please have a data cassette r
eady.":? "Press PLAY/RECORD keys when
the notes":? "sound, then press RETURN
"
50 OPEN #1,8,0,"C:"
60 PRINT #1;A$;CHR$(155);B$;CHR$(155);
C;CHR$(155)
70 CLOSE #1
80 ? :? "Please press STOP key. Rewind
 the":? "cassette, then take it out."
90 END
```

Fig. 5.12. Recording variable values separately on a cassette.

pronounced 'hash one'. We are 'opening channel number one', meaning that we are preparing part of the memory of the Atari to store this data before it is delivered to the tape. The next part, 8, is a code that Atari recognises as meaning an output. The zero is there because this type of command needs two numbers between the #1 and the final "C:". Using a zero doesn't cause anything to happen, but it prevents an error message from being delivered! Finally, the "C:" means 'send the data to the program recorder'.

This prepares for the recording of data, and we can now get round to making the actual recordings. The instruction that directs the data to the right 'channel' is PRINT#1, and it must be followed by the name of the variable (or a list of variables, with commas between them) that we want to record. The important point is that the computer is just gathering the data together during this time. When the time comes for the computer to make the recording you will be reminded by the usual two honks that you must press the PLAY and RECORD keys on the cassette recorder. Line 7Ø contains a very important instruction, CLOSE#1. This is the one that makes sure there is nothing left to record, and 'closes the file'. If you omit this, only a small part of your data will be recorded, and it will not replay correctly. Omitting CLOSE is called 'leaving your files open'. You wouldn't want to be seen in that state, would you? The program ends with a message reminder that you need to press the STOP key of the recorder, and take the cassette out.

Now we have to see if we really have a recording here. I've illustrated the steps in Fig.5.13. Line 2Ø prints a reminder on the screen, and the warning honk that you'll get at line 3Ø will remind you

```
10 ? "}":DIM X$(5),Y$(50)
20 ? :? "Please prepare the data casse
tte for":? "replay. When the note soun
ds, press":? "PLAY key, then RETURN."
30 ? :OPEN #1,4,0,"C:"
40 INPUT #1,X$,Y$,Z
50 ? "X$ is ";X$:?
60 ? "Y$ is ";Y$:?
70 ? "Z  is ";Z
80 ? :?
90 ? "Please press STOP key of recorde
r."
```

Fig. 5.13. Replaying variable values from a cassette. The variable names can be different, but the variables must be of the correct type (a string value cannot be read into a number variable).

to press the PLAY key. Remember that the tape must have been wound back.

Line 3ϕ then starts the action of loading the array, using the instruction word OPEN again. This time, we want an input, so we use OPEN#1, followed by 4, this time to mean an input from cassette, and the zero is followed by the "C:" again so as to specify that we want to get the data from a cassette. The advantage of using this type of system is that you don't have to learn very much that's new when you upgrade to the use of disks. The only change, in fact, is the use of "D:", along with a 'filename' in place of "C:". Lines 5ϕ, 6ϕ, and 7ϕ then print the values of the variables that have been read from the tape. Note that this does not have to use the same names as the names that we used when we were recording. It's the values that are replayed, not the names! What you do have to watch is that you don't replay a string and try to give it a number variable name. The computer won't accept that. Line 9ϕ delivers a message to remind you to press the STOP key of the recorder, and then it's all over. Data recording is easy the Atari way, providing you know the rules. If you want to record and replay long strings, however, particularly with a disk system, then there are a lot more rules to learn. The program in Fig. 9.15 (Chapter 9) illustrates some of the techniques, but at this stage in learning to use your Atari, it's too early to worry about these points.

Chapter 6
Special Effects

Any modern computer is expected to be able to produce dazzling displays of colour and other special effects. The Atari 600XL is no exception, and in this chapter we'll start to look at some of the effects that are possible. The Atari range of computers has always been noted for the spectacular screen displays that they can produce. It's important to point out, however, that you can't expect to be able to produce dazzling effects while you have your L-plates on. For one thing, to produce some of the displays that you may have seen in Atari games, and have sampled in the self-check routines, requires a different type of programming. Instead of using the BASIC program language, a system of number codes called 'machine code' has to be used to get the best out of the display system. That is very definitely not beginner's work, and we won't look at it here. All that we'll do is to introduce some of the principles, enough to start you on the way to modest but useful colour pictures.

To start with, you have to know some of the terms that are used, and the first of these is *graphics*. 'Graphics' means pictures that can be drawn on the screen, and all modern computers have instructions that allow you to draw such patterns. In connection with these patterns, you'll see the words *low resolution* and *high resolution* used. 'Resolution' isn't such an easy term to explain. Imagine that you are creating pictures on a paper sheet of about 11 inches across by 8 inches deep – that's roughly the size of a TV screen that is described as being a 14-inch screen (it's about 14 inches diagonally!).

Now if you are asked to create the pictures by sticking rectangles of coloured paper on to the sheet, you are dealing with picture making in a way that is very similar to the way that the computer operates. Suppose that you are allowed only 240 pieces of paper, of such a size that all 240 will fill the screen. You couldn't draw very finely detailed

pictures with so few large pieces, and this is what we mean by low resolution. On the other hand, if you were provided with pieces so small that you would need 61440 of them to fill an entire screen size, you could produce very much more detailed pictures. This is what we mean by high resolution. The Atari 600XL has both low and high resolution graphics available, and the figures that I have used correspond to the size of the blocks that the 600XL can use. You have, in fact, a lot more choice than this, because the 600XL allows you sixteen different combinations of possibilities. Each of these is called a 'graphics mode'. In any mode, you get a fixed amount of resolution, a fixed number of colours that you can use, and the use of special graphics instructions. Why are there so many modes? It's quite simple, really. When you make anything appear on the screen of the TV receiver, you have to store a pattern in the memory of the computer. This is the only way that the screen can go on showing the same picture. When you change what is stored in the memory, you change what is shown on the screen. Part of the memory of the 600XL is reserved for this kind of use; it's called the 'display memory'. Now when you use the lowest possible resolution of 240 pieces, this uses only a small amount of memory, 420 bytes to be precise. This leaves you with 13898 bytes for your program. If you select the highest possible resolution, however, which needs 8138 bytes just to keep the screen display looking good, then you'll find yourself with just 6180 bytes of precious memory left for your program. These figures assume that you are using the standard 16K version of the 600XL. The 'K' is the unit of memory, and it means 1024 bytes. 16K is $16 \times 1024 = 16384$ bytes. You don't have all of these available to you, however, because the Atari needs some of them for its internal 'note-keeping' actions when it runs a program. If you have fitted the 64K Memory Module, then you don't have to be quite so careful about memory. The point is that these dazzling displays have to be paid for, and the payment is in memory. You can't have a very long and complicated program which also has an ultra-high resolution colour display – not with 16K of memory, anyway.

That's why there are so many modes for graphics. You can decide which mode will suit what you are going to do, find out how much memory it leaves you with, and make up your mind if it's going to be possible with 16K of memory. At least you have the choice – not all computers allow you the luxury of deciding how much you are going to use for screen displays. Figure 6.1 is a table which shows the

GR. Mode	Type	Columns	Row (Split)	Row (Full)	Colours	RAM Split	RAM Full	Remaining
0	Text	40	—	24	1+	—		13326
1	Text	20	20	24	5	13644		13646
2	Text	20	10	12	5	13894		13898
3	Graphics	40	20	24	4	13884		13886
4	Graphics	80	40	48	2	13624		13622
5	Graphics	80	40	48	4	13144		13142
6	Graphics	160	80	96	2	12144		12134
7	Graphics	160	80	96	4	10128		10018
8	Graphics	320	160	192	1+	6206		6180
9	Graphics	80	—	192	1	—		6180
10	Graphics	80	—	192	9	—		6180
11	Graphics	80	—	192	16	—		6180
12	Graphics	40	20	24	5	13164		13166
13	Graphics	40	10	12	5	13654		13658
14	Graphics	160	160	192	2	10048		10022
15	Graphics	160	160	192	4	6206		6180

Fig. 6.1. The Graphics Modes of the Atari. These are selected by GRAPHICS (or GR.) and a number. The chart shows the options that you have.

options that are available to you, and how much memory each of them leaves in a 16K machine. How do you find such information for yourself? Easy, just type ?FRE (Ø) and press RETURN, and the computer will show how much of the memory is unused. When this figure gets to 1000 or less, you are just about out of memory. You may be able to type some more instructions, but the program won't be able to RUN. The reason is that running a program takes more memory than just storing it.

Vivid impressions

The best place to start on our exploration of special effects is with Mode Ø. Now this is the 'everyday' working mode, and it's what you are provided with each time you switch on the 600XL. Take a look at the program in Fig. 6.2. This demonstrates the shapes that we can produce on the screen in this mode, by making use of code numbers. Now we haven't confined ourselves to the familiar (it should be by now!) ASCII range of code numbers, 32 to 127. Instead, we've gone for the full range from Ø to 255. Almost the full range, that is. We

have to omit one code, 125. That's because the instruction PRINT CHR$(125) will, in fact, clear the screen! If we include 125 in our list, then, we shall see the shapes for codes Ø to 124 for only a split second before the screen clears. The program of Fig. 6.2 filters out code 125,

```
10 ? "}":FOR N=0 TO 255
20 IF N=125 THEN 40
30 ? CHR$(N);" ";
40 NEXT N
50 END
```

Fig. 6.2. A program that demonstrates the characters in Mode Ø.

by using line 2Ø, but causes each other code to print its corresponding character. Now you'll see that a lot of these characters are familiar, just the letters of the alphabet, the digits of our numbers, and the punctuation marks. There are some other shapes, though. These are the graphics shapes. You get an assortment of these for code numbers Ø to 28 inclusive, all in light-on-dark. Codes 28, 29, 3Ø and 31 do not produce any *shapes* on the screen, but they still have an effect. CHR$(28) will move up by one line, CHR$(29) will cause a new line to be selected, CHR$(3Ø) will move back one space before printing, and CHR$(31) will move one space forward before printing. These codes are called 'cursor' control codes. The cursor, remember, shows where the next character will be printed on the screen. There's a puzzle here, because you will see that CHR$(28) has been printed as an up-arrow, and it hasn't caused the cursor to move up by a line. That's because it followed CHR$(27). 27 is the code for the ESC key, and it can cause changes in any code that is printed immediately after it. If you type on the keyboard ?CHR$(28) (then RETURN), and then try ?CHR$(27);CHR$(28), you'll see the difference. Similarly, if you try ?CHR$(27);CHR$(29), you will see the down-arrow printed, and you can print arrow shapes for codes 3Ø and 31 in this way also.

The codes don't end at 127, however. Starting at code 128, the shapes all repeat – but this time in inverse video. Instead of having light-on-dark, we have dark-on-light. I've used the words light and dark, rather than light-blue and dark-blue deliberately because, as we shall see shortly, we can change the colours that we use quite easily. Once again, of these codes from 128 on, some will display a character only if CHR$(27) has been used just ahead of them. Codes 156 to 159, and 253 to 255 are of this type.

Atari chameleon

Now it's time to look at how we can change the colour of the whole screen, the background, and the characters on it. Graphics Mode ∅ allows you only one screen colour at a time, with the characters printed in lighter shades than the background.

Though there's only one colour, we still have a lot of control over the appearance of the display. The reason is that we can select what the colour will be, and we can also select how bright it will be. In addition, we can change the colour of the border that surrounds the main (display) part of the screen. All of this is done with a command called SETCOLOR. When you use this, remember the American spelling of 'COLOR' – the command won't work if you spell it 'SETCOLOUR'. We'll have to take a look at this important command because, of all the BASIC commands we have used so far, this is the one which is the hardest to understand. It's also one which will cause the ooohs and aahs from your friends when you use it correctly!

SETCOLOR has to be followed by three numbers, separated from each other by commas. The first of these numbers is called a 'register' number. A 'register' in this sense means a special controller inside the 600XL, and the number that is used will control some feature of the display, since there is a register number for each feature. The registers are numbered ∅ to 4 (that's five of them). Register number 2 controls the screen colours in Graphics Mode ∅, so for the next few examples, we shall use a 2 immediately following the SETCOLOR command. The other two numbers are for colour and brightness. The colour

Colour No.	Colour	Colour No.	Colour
0	Grey (Black at zero luminance)	8	Dark blue
1	Gold	9	Green-blue
2	Orange	10	Blue
3	Red	11	Dark blue
4	Pink	12	Green
5	Violet	13	Dark green
6	Purple	14	Olive-green
7	Light blue	15	Orange

Fig. 6.3. A table of colour numbers.

control number can use a value of ∅ to 15, and each of these numbers will cause a different colour to be selected. Figure 6.3 shows a table of numbers and colours. Some of these colours are hard to describe, and what you actually see depends quite a lot on how well you have tuned the TV receiver. Better colour displays can always be obtained with the use of a colour monitor. This looks like a TV set, but it can be connected to the 600XL more directly, using the Atari's colour signals direct instead of having them transmitted into the aerial socket. This produces much clearer colours, free of the fuzziness and wavy lines that you often see on colour TV receivers. A colour monitor is rather a luxury item unless you are a dedicated computer nut, but it's worth remembering that a colour monitor will also give much better pictures with a video recorder. What it can't do (unless you buy a special monitor/receiver) is receive TV pictures from an aerial.

Back to the pictures. When you have assigned a number for colour, you can then assign the third number of SETCOLOR. This controls the background brightness or 'luminance', as the manual calls it. The numbers you can use here are the even numbers from ∅ to 14. You can use odd numbers if you like, but an odd number won't produce any different amount of brightness. It will give the same brightness value as the next lower even number. If you use brightness value of 7, for example, it will give the same brightness as a value of 6. Perhaps it would be a good idea if we take a look at what these colours and brightnesses look like on the screen. Figure 6.4 is a program which selects the colours one by one, and then runs through the range of

```
10 GRAPHICS 0:FOR COL=0 TO 15
20 POSITION 6,3:? "Colour ";COL
30 FOR LUM=0 TO 14 STEP 2
40 SETCOLOR 2,COL,LUM
50 POSITION 6,LUM+5:? "Brightness is "
;LUM;" now."
60 FOR J=1 TO 1000:NEXT J
70 NEXT LUM:? "}":NEXT COL
80 ? "END.":END
90 GRAPHICS 0:FOR LUM=0 TO 14 STEP 2
100 SETCOLOR 1,9,LUM
110 ? "This is character brightness ";
LUM
120 FOR J=1 TO 1000:NEXT J
130 NEXT LUM
140 END
```

Fig. 6.4. A program to demonstrate background and foreground colours.

background brightness values. Note that when the brightness of the background is the same as the brightness of the characters, the characters are invisible. This is a simple way of hiding something that is on the screen. By a change of brightness, you can then suddenly reveal all! When you switch on the Atari, the settings of colour and brightness are colour 9 and background brightness 4. The characters (or foreground) are printed in luminance 1Ø, as the progam demonstrates. These are called the 'default' settings for Mode Ø, and in each other graphics mode you will find that you can get prearranged values of colour(s) and brightness like this. You can change them if you want to, but you will still have a good range of colours even if you don't make any use of the SETCOLOR command. Lines 1ØØ to 13Ø show how the brightness of the characters can be changed by using register 1.

You can also control the colour of the border around the picture. This is normally black, but it can be altered, as usual, by SETCOLOR. For the GRAPHICS Ø mode, register 4 controls the border colour, and we can use exactly the same range of colour and brightness numbers as we used for the main screen display. Figure 6.5

```
10 GRAPHICS 0:FOR SCOL=0 TO 15:SETCOLO
R 2,SCOL,4
20 POSITION 9,10:? "THIS IS YOUR TEXT!
"
30 FOR N=1 TO 20
40 SETCOLOR 1,SCOL,2
50 FOR J=0 TO 15:SETCOLOR 4,J,8:FOR X=
1 TO 30:NEXT X:NEXT J
60 SETCOLOR 1,SCOL,12
70 FOR X=1 TO 200:NEXT X
80 NEXT N
90 GRAPHICS 0
100 NEXT SCOL
```

Fig. 6.5. Illustrating your choice of border colours.

illustrates the border colours, and also shows how changes of border and screen colour can bring a bit of interest even to a simple display of text. You may find that on an ordinary colour TV the picture size changes slightly as the background colour and brightness change. A colour monitor should be free of this defect. TV receivers are designed to make the TV commercials clear, not to give you sparkling computer displays!

Placing characters

Up to now, we have produced text on the screen by using the PRINT

instruction. There are two ways of using PRINT. Taking the letter 'A' as an example, we could use PRINT"A" or PRINT CHR$(65). We can also use PRINT to produce, direct from the keyboard, characters that don't even appear on the keys! This is done by holding down the CONTROL key at the same time as you press one of the letter keys. The appearance of a keyboard on which these shape characters are printed is shown on page 6 of the Atari 600XL booklet. There is yet another way in which we can produce a letter on the screen, however. This makes use of two more commands, COLOR and PLOT.

Mode zero uses COLOR in a way that is rather misleading until you get used to it. COLOR, as far as Mode Ø is concerned, is simply another kind of PRINT command. It has to be followed by a number which is the ASCII code number for a character, but you don't need to use CHR$. For example, COLOR 65 should produce the same effect as PRINT CHR$(65). It's a lot less to type, to start with! Using COLOR by itself, however, doesn't produce anything on the screen. Nothing appears on the screen until you use PLOT. PLOT specifies where the character is to be placed, and it has to be followed by two numbers. These are the same numbers as you would use with POSITION. The first number is the column number, and we usually call it the X number when we use PLOT. The second number is the row number, called the Y number. X and Y are used rather in the same way as they are for graph plotting, except that the Y number is always measured from the top of the screen. For example, PLOT2Ø,5 would mean placing a character at position 2Ø across the screen (in which the left-hand side is position 2, and the right-hand side is position 38), and in row five (which is the sixth from the top, because the top row is numbered Ø).

Confused? Try an example program, which places letters and shapes, picked at random, all over the screen. Figure 6.6 shows the program. It starts with GRAPHICS Ø (which you can type as GR>Ø). This clears the screen, and sets zero mode graphics, just in case you were using another mode previously. The character code

```
10 GRAPHICS 0:FOR N=1 TO 500
20 CH=INT(RND(0)*256):IF CH=125 THEN 2
0
30 X=INT(RND(0)*38)+2
40 Y=INT(RND(0)*24)
50 PLOT X,Y:COLOR CH
60 NEXT N
```

Fig. 6.6. Using COLOUR and PLOT to place characters on the screen.

numbers are chosen at random, using the method that you have met previously. We have to exclude 125, however, because that would clear the screen. This is done in line 2Ø. We also pick random position numbers. For Mode Ø, the X numbers have to be in the range 2 to 39, and the Y numbers have to be in the range Ø to 23. Once all three numbers are picked, we can put the character in place by using COLOR and PLOT. Would you like to try wide-screen operation? You can widen the part of the screen that you use, so that you can use X numbers of Ø to 39, by typing:

```
POKE82,Ø
```

POKE is a command that acts directly on the memory of the Atari, and it's one that we shall try to avoid in this book as far as possible. This is because if you make a mistake in a number that you use with POKE, you can cause strange effects which you won't be able to undo except by switching off and starting again. You'll find a lot more information about the use of POKE in the more advanced books on Atari BASIC.

Image building

There are 29 shapes that are available for use in picture building in Mode Ø. The best way to go about building patterns with them is to imagine that the patterns are constructed by filling in squares on a 3 × 3 grid. Figure 6.7 shows this grid, whose overall size is the size of a character. Not all of the shapes fit this grid perfectly, but most of the ones that we will need for pictures do. To make a picture with these units (or pixels), we need a larger-scale plotting grid.

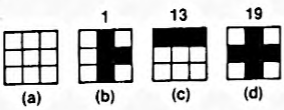


Fig. 6.7. (a) The 3 x 3 grid on which a lot of the GR.Ø characters fit. This is for planning purposes. Characters 1, 13 and 19 are illustrated in (b), (c) and (d).

Figure 6.8 shows a grid on which you can plan multiple-pixel shapes. For the less gifted (me especially) a bit of planning is needed. The best planning aids that you can invest in are a ruler, a 2B pencil, and a pad of tracing paper. Start by sketching out the pattern that you want to use, remembering to shade in only the blocks of the 3 × 3 portions, as Fig. 6.9 shows. This is a fairly elaborate shape, and I've

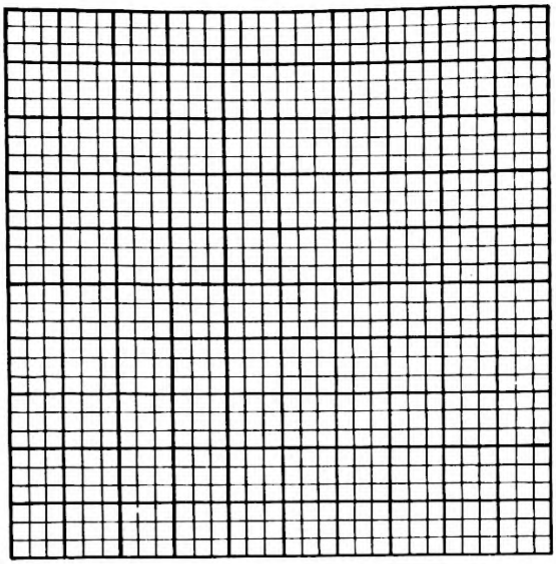


Fig. 6.8. A grid for planning shapes that are made out of a number of characters.

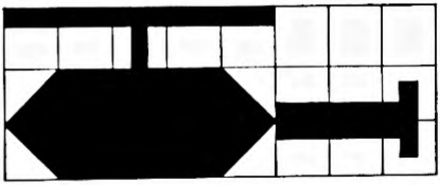


Fig. 6.9. An example of a shape created from characters in Mode ∅.

used it to show just how effective these simple graphics shapes can be. Planning is important, because you will save a lot of time if you can enter the shape into the program in one go.

Yes, you're right – I shall have to give an example. Start with Fig. 6.9, which shows a 'helicopter' type of shape. This uses 21 shaded blocks and 3 empty blocks to establish the shape, and our task is to program this shape on to the screen. The job starts with writing the shape codes for each block. We get these by looking at the shapes and code numbers that are shown later in Fig. 6.15. This is a chart of shapes for Modes 1 and 2, but the same set of characters is used for Mode ∅. The codes ∅ to 31 in Mode ∅ give the shapes that are listed as 'alternate' in Fig. 6.15. We then have to take the shapes that are closest to what we need – some patterns that we draw may not fit exactly to the shapes that are available. Remember that you have two

```
10 GRAPHICS 0:POKE 752,1:FOR ROW=1 TO
3
20 FOR ACR=1 TO 8
30 READ D
40 COLOR D:PLOT ACR+6,ROW+5
50 NEXT ACR:NEXT ROW
60 POKE 752,0:END
100 DATA 18,18,23,18,18,32,32,32
110 DATA 8,160,160,160,10,14,14,3
120 DATA 138,160,160,160,136,13,13,5
```

Fig. 6.10. A program to display the shape on the screen.

sets of shapes, the ordinary shapes codes Ø to 31, and the inverse-video shapes, codes 128 to 159. By writing down these numbers, we have completed the hard part of designing a graphics shape.

The next part is programming. Figure 6.10 shows how this shape could be programmed on to the screen. We print the first line starting at PLOT 7,6, and we take the code numbers for the shapes from a DATA line, using COLOR D to place the codes on the screen. Since the pattern consists of three rows, we need three loops to place all of the codes, and we need to use the codes for blank squares as well. The command, POKE 752, 1 has the effect of suppressing the cursor, and POKE 752,Ø restores it. This way, we don't have the cursor upsetting the appearance of the picture. Try deleting the POKE instruction if you doubt it!

Stringing it up

The method that we used in Fig. 6.10 is perfectly satisfactory for creating shapes that are printed only in one place. We can, in fact, create some very useful shapes in this way. It is much more interesting, however, to be able to print a shape easily wherever we want it. An alternative method is to make a single string out of several lines of graphics characters. Placing the shape on the screen then amounts to printing a single string, a quicker and easier method. What makes it possible is that we can include the cursor control characters in such a string, so that the printing can move up, down or across as we please. This avoids having to plot empty spaces.

Let's look at it in action. So that you can see what is going on, we'll use the same data, creating the same shape. Figure 6.11 shows the program that we need. Lines 1Ø to 2Ø clear the screen, dimension variables, and then pack a string SP$ with cursor control characters. These are characters that don't make any mark on the screen, but

```
10 GRAPHICS 0:POKE 752,1:DIM A$(65)
20 DIM SP$(15):SP$=CHR$(30):SP$(8)=SP$
:SP$(2)=SP$:SP$(1,1)=CHR$(29)
40 FOR Y=1 TO 3:FOR N=1 TO 8
50 READ D:A$(LEN(A$)+1)=CHR$(D)
60 NEXT N:A$(LEN(A$)+1)=SP$
70 NEXT Y
80 POSITION 10,5:? A$
90 ? :? :POKE 752,0
100 END
110 DATA 18,18,23,18,18,32,32,32
120 DATA 8,160,160,160,10,14,14,3
130 DATA 138,160,160,160,136,13,13,5
```

Fig. 6.11. Making a single string of a picture. The string SP$ contains a 'next line' character, and several 'backspace' characters. This is what causes the other lines of characters to be printed in the correct places.

shift the cursor. CHR$(3Ø), for example, moves the cursor left, and CHR$(29) moves it one line down. The effect of PRINT SP$, then, will be to move the cursor down, and move it left to the start of the line again. Lines 4Ø to 7Ø then create a string A$ which contains the shapes. After each line has been read, the string SP$ is added so as to place the cursor where it is needed on the next line. This is how the whole set of actions can be placed in one string. A$ is then the helicopter shape! The shape can then be printed in any place on the screen.

Shaping up for games

We now have some useful programming methods available for games use. At this point, I think it's necessary to point out that a home computer, programmed in BASIC, can't produce the dazzling effects that you see on arcade games machines. For one thing, the arcade machines are not programmed in BASIC, and for another, they cost many times as much as a home computer. Most of them will earn enough in takings in a night to buy dozens of home computers! That doesn't mean that you can't play good games on home computers. It does mean, though, that you can forget about designing lightning

```
10 GRAPHICS 0:POSITION 2,11
20 DEG :COLOR 46:POKE 752,1
30 FOR X=2 TO 39
40 PLOT X,10+SIN(10*X)*10
50 NEXT X
60 POKE 752,0
```

Fig. 6.12. A graph-drawing program, making use of PLOT. Notice how DEG is used so that angles measured in degrees are used.

action games, with all sorts of shapes moving fast in different directions. Forget them, that is, until you know much more about Atari programming. You have in the Atari 600XL, however, a computer that is capable of the sort of effects that you may have seen and marvelled at. All you need is experience.

Graph plotting

The PLOT instruction of your Atari, along with the code for a white dot, can be used for graph plotting. Figure 6.12 shows a graph-drawing program. This draws a graph of what is called the 'sine' of an angle, showing how the value of this quantity changes as the size of the angle changes. Line 3Ø starts the loop which makes use of all the permitted values of X. The graph shapes are achieved by using the SIN function. This by itself draws a wavy shape, and that's what is drawn by line 4Ø. The multiplying factors are put in to make the shape fill a reasonable amount of the screen in the Y direction. The sine of an angle cannot have a value less than −1 or more than +1, so we have to 'amplify' it a bit by multiplying by 1Ø. The value of X has to be multiplied by 1Ø to make the range of angles suitable. The Atari 600XL will use angles in units of degrees only if you type DEG. If you forget to use DEG before you start using angle sizes, the 600XL will use a more natural unit, the radian. One radian is about 57 degrees. To return to using radians after using degrees, you need to have the command RAD.

We can't leave this topic, however, without looking at another instruction which operates with the same numbers as PLOT and POSITION. This is LOCATE, and it also has to be followed by the now-familiar X and Y numbers, and also by a number variable. LOCATE does not make anything appear on the screen. On the contrary, it reports back to your program what is present at a selected position on the screen. Suppose you have the instruction:

```
LOCATE 2Ø,5,Q
```

Now if this position is empty, the value of Q will be 32. If the position contains a character, the value of Q will be the code for that character when we are using Mode Ø.

We can show this in action in a 'bouncing-ball' type of program. Figure 6.13 shows how this is done. Lines 1Ø to 3Ø create two vertical lines on the screen. Line 4Ø then establishes starting values for three

```
10 GRAPHICS 0:POKE 752,1:FOR Y=0 TO 23
20 COLOR 22:PLOT 2,Y:NEXT Y:COLOR 2
30 FOR Y=0 TO 23:PLOT 38,Y:NEXT Y
40 K=1:X=21:Y=0
50 GOSUB 1000
60 LOCATE X+K,Y,Q
70 IF Q<>32 THEN K=-K:Y=Y+1
80 IF Y<23 THEN 50
90 POKE 752,0:END
1000 COLOR 20:PLOT X,Y
1010 FOR J=1 TO 10:NEXT J
1020 COLOR 32:PLOT X,Y
1030 FOR J=1 TO 10:NEXT J
1040 X=X+K:RETURN
```

Fig. 6.13. A 'bouncing ball' program that makes use of LOCATE.

variables. The variables X and Y will be used to control the position of a block, and they are chosen to give a starting position near the top centre of the screen. A loop now starts in line 5ø. This line calls a subroutine that prints a block at the X,Y position, pauses, deletes the block, and then adds the value of variable K to X. Line 6ø then tests this position, to see if one more step in the X direction would be a 'wall' character, one which contains a character which is not 32. If the next position is blank, nothing is done in line 7ø. If, however, the next position is a 'wall', then the Y number is increased by 1, and the change of X is reversed by reversing the sign of K. If K is positive, then K = −K will make it negative. If K is negative, then K = −K will make it positive.

Line 8ø then tests to find if the value of Y corresponds to the last line on the screen, and loops back to line 5ø if this is not so. Because the value of X + K is tested, the 'ball' never actually hits the 'wall'! The overall effect is of a bouncing block, which gradually moves down the screen as it bounces. By altering the order of these instructions, you can cause a clean bounce, leaving no trace, or you can knock holes in the wall each time the block hits it! There's a lot of enjoyment to be had from this simple set of instructions.

Modes 1 and 2

All of the effort you have put into graphics so far has been concerned with Mode ø. There are two other modes which are classed as 'text modes', like Mode ø, but which are rather different. They are so similar that we'll deal with them together.

We'll do the examples with the more useful one, Mode 1. This is

programmed by starting your graphics program with the instruction GRAPHICS 1, or its abbreviation, GR.1. When you carry this out, you'll see a most remarkable change in the screen. The top twenty lines of the screen are now reserved for graphics, and text will only appear in the lowest four lines! This is called 'split-screen operation', and it's a feature of nearly all the Atari graphics modes, with the exception of modes Ø,9,1Ø and 11.

Now you're not stuck with this split screen if you don't want it. If, when you use the GRAPHICS instruction, you add 16 to the number, the screen will *not* be split. In Mode 1, you will then have a screen of 2Ø columns (across) and 24 rows (down). The snag is that you can't LIST a program or print text in the usual way on this 'full screen'. When your program ends or causes an error message, the Mode will change back to Mode Ø. When you use a 'full-screen', then, you have to make sure that messages like 'READY' don't appear. This is done by having an endless loop, a line such as:

```
5ØØ GOTO 5ØØ
```

which will make sure that the computer cannot return to Mode Ø until you press the BREAK key.

The colourful choice

The point of having Modes 1 and 2 is that they permit a much larger range of colours to be seen on the screen at a time. If we ignore the border colour, then both of these modes allow up to 4 colours for the characters that we print. What we need to know now is how we select the characters and how we select the colours. Like many other Atari procedures, it looks complicated until you understand the principles, so here goes.

To start with, we can select the range of colours that we want. We can pick any four (I'm ignoring the fifth border colour for the moment) by using SETCOLOR. If you don't use SETCOLOR, you will have the 'default' colours of Orange, Green, Dark Blue, and Red, with the border black. You may be perfectly happy to use these colours, but if you want to change them, or their brightness values, you can. The colours are changed by SETCOLOR, using register numbers Ø to 3. Figure 6.14 shows the default colours and their brightness values. If you want to change the Register 1 colour from

Register	Colour No.	Brightness	Colour
0	2	8	Orange
1	12	10	Green
2	9	4	Dark blue
3	4	6	Red
4	0	0	Black

Fig. 6.14. The 'default' register colours. These are the colours that you will use if you don't change them with SETCOLOR.

Green to Purple, for example, and with high brightness, then you need to type:

```
SETCOLOR 1,5,12
```

and the other registers can be changed in the same way, just as you please.

The next item is to produce characters. We can use PRINT, but with a difference! PRINT, used in the normal way, will produce text on the 'text screen', that is, on the lowest four lines. If you are using the whole screen, then a PRINT instruction will cause the whole display to return to Mode Ø! To print on the graphics screen in Modes 1 and 2, you have to use PRINT#6 (pronounce it as PRINT-HASH-SIX). Try this – type GR.1, which is the short version of GRAPHICS 1, and press RETURN. You'll see the screen split, and the READY will appear in blue at the foot of the screen. Now type: PRINT#6;"TEST", and press RETURN. You will see the word TEST appear at the top left-hand side of the screen, in colour orange, and your instruction, along with 'READY' at the foot of the screen in blue. If you try PRINT#6,TEST, you will see the word appear on the right-hand side of the screen this time, because of the action of the comma. In Modes 1 and 2, a comma causes text to be placed starting at the centre of the screen. What you cannot do, however, is to type PRINT#6"TEST" without any punctuation mark, because this causes an error message.

Getting the characters

You can create text and shapes in Modes 1 and 2 in just the same way as you can in Mode Ø, by using PLOT and COLOR. This time,

Value for Colour Register				Character		Value for Colour Register				Character	
				Standard	Alternative					Standard	Alternative
0	1	2	3			0	1	2	3		
32	0	160	128			64	96	192	224	@	
33	1	161	129	!		65	97	193	225	A	a
34	2	162	130	"		66	98	194	226	B	b
35	3	163	131	#		67	99	195	227	C	c
36	4	164	132	$		68	100	196	228	D	d
37	5	165	133	%		69	101	197	229	E	e
38	6	166	134	&		70	102	198	230	F	f
39	7	167	135	'		71	103	199	231	G	g
40	8	168	136	(		72	104	200	232	H	h
41	9	169	137	)		73	105	201	233	I	i
42	10	170	138	*		74	106	202	234	J	j
43	11	171	139	+		75	107	203	235	K	k
44	12	172	140	,		76	108	204	236	L	l
45	13	173	141	-		77	109	205	237	M	m
46	14	174	142	.		78	110	206	238	N	n
47	15	175	143	/		79	111	207	239	O	o
48	16	176	144	0		80	112	208	240	P	p
49	17	177	145	1		81	113	209	241	Q	q
50	18	178	146	2		82	114	210	242	R	r
51	19	179	147	3		83	115	211	243	S	s
52	20	180	148	4		84	116	212	244	T	t
53	21	181	149	5		85	117	213	245	U	u
54	22	182	150	6		86	118	214	246	V	v
55	23	183	151	7		87	119	215	247	W	w
56	24	184	152	8		88	120	216	248	X	x
57	25	185	153	9		89	121	217	249	Y	y
58	26	186	154	:		90	122	218	250	Z	z
59	27	187	None	;		91	123	219	251	[	
60	28	188	156	<		92	124	220	252	\	
61	29	189	157	=		93	None	221	253	]	
62	30	190	158	>		94	126	222	254	^	
63	31	191	159	?		95	127	223	255	_	

Fig. 6.15. Characters, colours and registers for Modes 1 and 2. You pick a character, and decide which colour to use by picking its code number!

though, you'll find that the character numbers have to be different! The characters are grouped into a main set of 64, and an alternative set. Now if you use the numbers 32 to 95, you get the characters printed in orange. This is also what you get when you use PRINT#6; followed by text. The 'ordinary' set of characters consists of the digits Ø to 9, the punctuation marks, and the upper-case (capital) letters.

The 'alternate' set consists of the graphics shapes that we have used already, along with the lower-case (small) letters. You have to choose whether to use ordinary or alternate characters, and you can't mix them. To switch over to the alternate set, you have to use: POKE 756, 226, and to switch back again, you have to use POKE 756, 224.

Now let's see how we produce coloured characters. Figure 6.15 shows a chart of characters and registers for Modes 1 and 2. A code value of 32 to 95 makes use of register 0, which means that the characters will be printed in the colour that is assigned to this register. If you don't change it, that's orange. Using a number between 0 and 31, or 96 to 127 gives the same shapes, but using register 1. If you haven't changed this with SETCOLOR, that gives green. The same applies to the other two registers.

```
10 GRAPHICS 1
20 GOSUB 1000
30 GRAPHICS 2
40 GOSUB 1000
50 END
1000 FOR N=1 TO 5
1010 READ D:COLOR D
1020 PLOT 5+N,5:NEXT N
1030 RESTORE :FOR Q=0 TO 15
1040 IF PEEK(756)=226 THEN POKE 756,22
4:GOTO 1050
1045 IF PEEK(756)=224 THEN POKE 756,22
6
1050 GOSUB 2000:SETCOLOR 4,Q,6
1060 FOR REG=0 TO 3:COL=INT(RND(0)*16)
1070 SETCOLOR REG,COL,12
1080 NEXT REG
1090 NEXT Q
1110 GOSUB 2000
1120 RETURN
2000 FOR J=1 TO 200:NEXT J
2010 RETURN
5000 DATA 65,116,193,242,73
```

Fig. 6.16. Illustrating the appearance of characters in Modes 1 and 2.

Figure 6.16 wraps all of this work up into one program. It starts in Mode 1, and prints a message, with the letters in different colours. It then changes the letters to shapes, by making use of the alternate set, does a bit of fancy colour changing with SETCOLOR, and then repeats the whole process in Mode 2. It's spectacular compared to what you can achieve in Mode 0, and it gives you a slight inkling of what can be done in these two modes! Fasten your safety-belt, there's a lot more to come.

Chapter 7
High Resolution Graphics

High resolution graphics means picture patterns that are created with smaller units than are possible with the text-size characters. The Atari allows you to use high resolution graphics instructions which operate with a variety of different screen plottings. These plottings can range from a rather low resolution 40 by 10 to a very high resolution 32 by 192. The elements of these high resolution grids are called *pixels*, and in the high resolution modes like 6,7,8,14 and 15, the pixels in this grid are much smaller than those we have used previously. With their use comes a new set of instructions that apply only to these high resolution pixels. When we work with the high resolution graphics of the Atari, we have to use much more of the memory of the Atari to store information about the pixels. Because the amount of memory that is normally used for placing text on the screen is no longer adequate, we can't mix normal text with high resolution graphics on the same part of the screen. We can, however, make use of the split-screen facilities of the Atari to place graphics on about five sixths of the screen, and text on the remainder.

The new commands

The use of any of the 'graphics modes', numbers 3 to 15 inclusive, allows you to use a special instruction that applies only in these modes. DRAWTO is its name, and its use is to draw patterns on any of the high resolution graphics screens. We normally use DRAWTO along with a command we have met before, PLOT. PLOT is used to place a dot, and DRAWTO is used to make a straight line from the PLOT position to another position. This is the pair of commands that can create all the fascinating patterns that you see in programs that you can buy.

Decisions, decisions

The Atari allows you a lot of choice about the use of high resolution graphics, and when you first encounter these choices, you may feel rather bewildered. Fear not, because I'll guide you through the confusion, starting now. There are thirteen different sets, or 'modes', of high resolution graphics, which are numbered 3 to 15. The differences between them are the size of the pixels, the number of colours that you can use, and the amount of memory that they need. Figure 7.1 shows the choices. You can see from the sizes of some of

Mode	Columns	Rows – Split	Rows – full	Colours	Pixel size (relative)
3	40	20	24	4	
4	80	40	48	2	
5	80	40	48	4	
6	160	80	96	2	
7	160	80	96	4	
8	320	160	192	1+	
9	80	–	192	1	
10	80	–	192	9	
11	80	–	192	16	
12	40	20	24	5	
13	40	10	12	5	
14	160	160	192	2	
15	160	160	192	4	

Fig. 7.1. The graphics only modes. Some of these are high resolution, others are not.

the pixels that some modes are not exactly high resolution. The selection is made by using the instruction GRAPHICS (or GR.), followed by a number. If you select GRAPHICS 3, for example, you will use one of the largest of the high resolution pixels, indicated by the pattern in Fig. 7.1. You will have the choice of four colours, but by the use of SETCOLOR, you can choose which of the sixteen colours of the Atari 600XL you want to make use of. Typing GRAPHICS 12 gives you the same size of pixel, but a greater choice of colours. You pay for this by using more memory, almost three times as much. Modes 9, 10 and 11 allow only full-screen operation; the others permit split-screen or full-screen as you want.

Before we can start on a high resolution graphics drawing, then, we need to make some choices. The GRAPHICS number is the first one, and it will help you to decide on some of the others. Once we have sorted this out, we can decide how we plan the drawing. As always, we have to plan these drawings, but there is a big difference between planning on a 40 × 10 grid and planning on a 320 × 192 grid! There is also a new use for the instruction word COLOR. In these graphics modes, COLOR selects which of the colour registers will be used for the PLOT and DRAWTO commands. In the language of computer graphics, it decides the current foreground colour. If, for example, we have in a four-colour mode, COLOR 2 placed just before a PLOT and a DRAWTO, then whatever is drawn will be in the colour of register 1. *Not*, you note, *colour 2*, but whatever colour has been assigned to register 1. Yes, I know it looks ridiculous, but that's the way it is. Figure 7.2 shows how COLOR is related to the use of registers in different modes. COLOR 0 is *always* the background colour. In the four-colour modes (3, 5 and 7) this is the colour of register 4. For the other, foreground, colours, we use COLOR numbers which are one more than the register number. In modes 3, 5 and 7, for example, COLOR 1 takes its colour from register 0, COLOR 2 takes its colour from register 1, and COLOR 3 takes its colour from register 2. In modes 4 and 6, COLOR 1 is foreground (register 0), and COLOR 0 is background (register 4). In mode 8, COLOR 1 uses register 1 (it just had to be different), and COLOR 0 uses register 2. The modes 9 to 15 use COLOR in even more eccentric ways. Getting back to the example, where register 1 has been used, the default colour for this register is green, but you can, as always, use SETCOLOR to assign any other colour that you like to this register. Clearing the screen? That's easy – though the CLEAR key doesn't

(a) How the registers are used.

Graphics Mode	Colour Register ∅	1	2	3	4
0	—	lum.	B/F	—	Br.
1	Ch	Ch	Ch	Ch	B/Br.
2	Ch	Ch	Ch	Ch	B/Br.
3	P/D	P/D	P/D	—	P/D,B,Br.
4	P/D	—	—	—	P/D,B,Br.
5	P/D	P/D	P/D	—	P/D,B,Br.
6	P/D	—	—	—	P/D,B,Br.
7	P/D	P/D	P/D	—	P/D,B,Br.
8	—	lum.	B	—	Br.

Notes: lum. - controls brightness only.
B - background colour.
B/F - background and foreground same colour.
Br. - border colour.
Ch - prints a character in foreground colour.
F - foreground colour.
P/D - foreground colour for PLOT or DRAWTO.

(b) The COLOR number and Register number

Register number	COLOR No. in Modes 3,5,7	4,6	8
0	1	1	—
1	2	—	1
2	3	—	0
3	—	—	—
4	0	0	—

Note: Register 3 is used only by Modes 0,1 and 2.

Fig. 7.2. How COLOR is related to SETCOLOR, and to what you see on the screen. This is the instruction that beginners to the Atari find the most puzzling.

work when the graphics modes are in use, and change of graphics mode automatically clears the screen. At times, this can be an unwanted nuisance, so if you add 32 to the graphics number, you'll find that the screen is not cleared! It's all good Atari planning to make it possible for you to create really interesting graphics. Yes, an

example would make it a lot clearer, but we have to know how to draw lines first.

Drawing the Atari line

DRAWTO is the new high resolution graphics instructions that we have to look at. DRAWTO will cause a straight line to be drawn between two points, and we have to specify what two points we want. This is done, as you might suspect after reading Chapter 6, by using X and Y co-ordinate numbers. The starting point for a line is obtained by using PLOT, which places a dot at a point. The position of the point is specified by the X and Y numbers that we use. These in turn, are decided by which mode we have selected. For example, if we have selected Mode 4, we can use numbers from Ø to 79 for X, and from Ø to 39 for Y (split screen) or Ø to 47 if we use the full screen.

Once the PLOT command has been used to position a dot, we can use DRAWTO. DRAWTO is also followed by X and Y numbers, in the same range as we used for PLOT. When the instruction is carried out, a straight line will be drawn from the point that was placed by PLOT to the new X and Y position that was used in the DRAWTO instruction. The line will be drawn in the 'current register colour', meaning the colour in the register that was assigned by COLOR. If you are making a drawing that consists of several straight lines, each joined to the one before, then you don't have to use PLOT again. Instead, you can have another DRAWTO instruction. This one will draw a straight line from the end of the *previous* line to the new point that is dictated by the X and Y values that you have used in the new DRAWTO instruction.

It seems reasonable, when you take it slowly, but there's a small hazard that Atari don't tell you about! Your drawing may not be visible! Unless you have specified a colour register, using COLOR, which is different from the background colour, you may find yourself drawing invisible lines. It's not so awkward as you might think. The advantage of this system is that you can make a drawing appear and disappear very rapidly. Take a look, for example, at the pattern in Fig. 7.3. This is a simple star pattern which, when it is first drawn, is visible because of the use of COLOR 2. After a short delay, we then make it invisible by using SETCOLOR 1,Ø,Ø. This instruction causes the register that COLOR 2 has specified to be of background colour and brightness, so the drawing disappears. After another short delay,

```
10 GRAPHICS 7:COLOR 2
20 PLOT 79,0
30 DRAWTO 2,39:DRAWTO 79,79
40 DRAWTO 157,39:DRAWTO 79,0
50 FOR J=1 TO 1000:NEXT J
60 SETCOLOR 1,0,0
70 FOR J=1 TO 1000:NEXT J
80 SETCOLOR 1,12,10
90 END
```

Fig. 7.3. Drawing a star pattern and making it disappear. This is done very simply by using SETCOLOR. Some computers have to 'undraw' the whole pattern!

the drawing appears again. The appearance can be very rapid, because you don't have to wait for the drawing actions, just for the SETCOLOR instruction. Note that COLOR 2 uses Register 1 in this Mode.

Planning the lines

How do you set about drawing with the PLOT and DRAWTO instructions, then? As always, planning is the key to successful drawing, so we start with a planning grid. This can be anything from 40 × 10 to 320 × 192, depending on which of the graphics modes you want to use. The simplest and best way to make your plans is to use ordinary graph paper. It should be scaled with 1, 5 and 10 mm lines, and it's ideal for Atari patterns. You'll find general-purpose graph paper like this is a lot cheaper than some of the 'special planning pads' that are sold in computer shops. How you go from there depends on which graphics mode you are using. Suppose, for example, that you want to create a pattern in Mode 6. This uses a 160 × 80 grid when split screen is in use, so you can draw a 160 × 80 mm rectangle on your graph paper. Now we should number from 0 to 159, and 0 to 78, and shade in the squares that we are going to light up. We can get away with something much simpler, though. If you number from 1 to 160, and 1 to 80, then this won't cause any trouble provided you don't take any line right to the edge of the paper. Also, it's easier to work with points rather than squares. You can then draw your patterns as lines on the rectangle that you have marked out. On this diagram, the X and Y numbers are then written at every place where a line changes direction. That's step 1 in planning a pattern. Step 2 is to choose a starting point. What you choose is a matter of convenience, and if you don't intend to move the pattern around, it doesn't much matter.

Take an example. Figure 7.4 shows a rocket shape. This is arranged

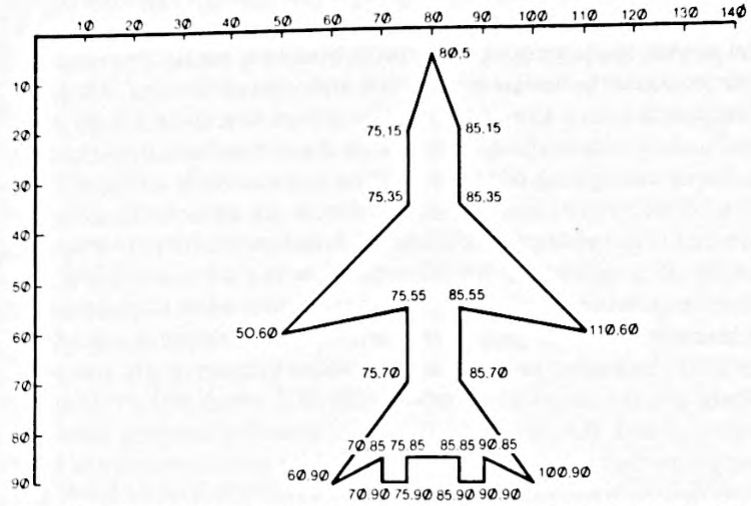


Fig. 7.4. A rocket shape which was drawn on graph paper. The X and Y numbers of the points have been written in.

well within a 160×96 rectangle, as used by Mode 6, full screen. I have written each X and Y number where the straight lines of the pattern change direction. We now need a starting point for the drawing, and 80,5 looks as good as any. A starting point of 80,5 means X=80, Y=5. Once this has been done, the rest is more or less plain sailing. We can write the program lines, as Fig. 7.5 illustrates. Line 10 has set up the mode, the colour set and the foreground/background colours. The drawing is done by reading X and Y values from a set of DATA lines, using a loop, because each new line starts from the end of the previous one. We have to prevent the full screen mode from disappearing by making an endless loop, as in line 40. You will need to press the BREAK key to return to normal service.

```
10 GRAPHICS 6+16:SETCOLOR 4,6,2:SETCOL
OR 0,3,12:COLOR 1
20 PLOT 80,5:FOR N=1 TO 21
30 READ X,Y:DRAWTO X,Y:NEXT N
40 GOTO 40
100 DATA 75,15,75,35,50,60,75,55,75,70
,60,90,70,85,70,90,75,90,75,85,85,85
110 DATA 85,90,90,90,90,85,100,90,85,7
0,85,55,110,60,85,35,85,15,80,5
```

Fig. 7.5. The program which will produce the rocket shape on the screen.

You might think that the next obvious step would be to illustrate animation of a pattern of this type. It would be, but not in this book! The reason is that the ordinary draw-and-wipe methods of animating patterns are very slow and clumsy. When you make use of these methods, you are always aware that the computer is drawing new patterns and wiping old ones because you can see it all happening. The Atari creates its animated patterns by an entirely different method. This makes use of what are called *sprites*, or *player-missile graphics*. A 'sprite' is a pattern which you can draw, and which can then be animated as you please. The animation is free from jerkiness, and is very convincing. The trouble is that to create sprites, you have to POKE number codes directly into the memory of the machine. There are no simple commands in BASIC which will create sprite patterns and then animate them for you. Fortunately, there are programs that will perform these actions for you. If you want to learn how sprites can be created and used, then take a look at my book, *Get More From The Atari*, which goes into a lot more detail about sprites and sprite graphics.

Demonstrating resolution

It's time to illustrate where we've got to so far. Figure 7.6 is a program that places dots on the screen. It uses Mode 11, which is unusual,

```
10 GRAPHICS 11:SETCOLOR 4,4,12
20 FOR N=1 TO 10
30 FOR Y=0 TO 191
40 X=INT(40*RND(0))
50 COL=INT(16*RND(0))
60 COLOR COL:PLOT X,Y
70 PLOT 79-X,Y
80 NEXT Y:NEXT N
90 GOTO 90
```

Fig. 7.6. Placing random dots on the screen, using Mode 11. This Mode uses a lot of memory – if you have a disk system connected to a 16K Atari, you can't run this program!

because it has fairly high resolution (80 × 192) and yet allows sixteen possible colours. In this mode, the colour that is placed on the screen is controlled *directly* by the COLOR instruction. That means that the number which follows COLOR is *not* a register number but a colour number, 0 to 15. The register that is used is register 4, and it controls the brightness of the colours only.

The program creates dots at random on the screen. This is done like

a kaleidoscope, with a random number being generated. This random number is between Ø and 39, and it is used to place a dot, in a randomly selected colour on a point in the left-hand side of the top line of the screen. A dot is then placed in the right-hand side of this same line, using the same colour, but placed symmetrically. This means that the second dot should be as far from the right-hand side of the screen as the first one was from the left-hand side. This is done by using 79 − X as the X number for the second dot. If the first dot was at 5, then the seat 79 - 5 = 74. This is five squares from the right-hand side, just as the first dot was five squares from the left-hand side. By programming a loop for the number of lines, a dot will be placed in each line. The process is repeated ten times so as to make a complete pattern. This program, incidentally, requires a lot of memory just to maintain the screen display.

If you're not so keen on dotty patterns, how about some random lines? The program of Fig. 7.7 creates lines at random, using Mode 7,

```
10 GRAPHICS 7:COLOR 2:PLOT 79,39
20 FOR N=1 TO 50
30 X=INT(160*RND(0))
40 Y=INT(80*RND(0))
50 DRAWTO X,Y
60 NEXT N
70 FOR COL=0 TO 15:SETCOLOR 1,COL,10
80 FOR J=1 TO 500:NEXT J
90 NEXT COL
100 GOTO 70
```

Fig. 7.7. Drawing random lines in Mode 7. You can produce a 'modern masterpiece' in one second flat.

split screen. The resolution in this mode is 160×80, so we have to use numbers Ø to 159 for X and Ø to 79 for Y. We pick a starting point of 79,39, which is the centre of the screen. A PLOT to this point makes a dot appear, and from there we go on to draw lines. Fifty lines are drawn, using randomly selected end points, and with randomly selected colours. Looks impressive, doesn't it? To round it off, an endless loop goes round the SETCOLOR registers changing the colours to make it all the more interesting. Beats watching soap opera any day!

Filling in the colours

Among the cleverest tricks of the Atari's graphics is the ability to fill in a closed shape with colour. This is a very difficult action if you have

to program it for yourself, and it's not easy even for the designers of a computer. Because of that, you find that any fill command is either very slow in action, or it obeys rather complicated rules. The Atari colour filling command is a fairly fast acting one, but it can only be used if its rules are obeyed very precisely.

The command word is XIO. Unlike many of the command-words that you have met so far, XIO doesn't exactly remind you of what the command does. This is because it's a 'general in-out command', one that is used for a number of different purposes. Filling in an area with colour just happens to be one of these actions. The rules are as follows.

(1) You have to have a closed shape drawn on the screen. By a closed shape I mean that the last line must end at the first point, with no gaps. None of the lines should run along any edge of the screen. All of the lines that make up the pattern should have been drawn in the same colour. This will be the colour that is used for the filling action.

(2) Take a point at the extreme lower right-hand corner of your pattern, and PLOT this point.

(3) Now find the X and Y values of the point which is at the top right-hand corner of your pattern. Use DRAWTO to draw a line to this point.

(4) Find the X and Y values for the point at the top left-hand corner of your pattern. Use DRAWTO to draw a line to this point.

(5) Find the X and Y numbers for the bottom left-hand corner of your pattern. Use the POSITION command to place the cursor at this point.

(6) Execute the instruction

```
POKE 765,n:XIO 18,#6,Ø,Ø,"S:"
```

using for n the COLOR number that you have used for drawing the lines.

This sequence of operations will result in a perfect colour fill only if you have gone through all of these steps in order. As it's written, the idea is that it should be used with four-sided shapes, but if you make

two of the points very close together, the shape will appear to be a triangle, and the XIO instruction will still work. Where it becomes difficult is when you want to fill a complicated shape with colour. You then have to break the shape up (mentally) into pieces which are four-sided or triangular, and fill each of them separately.

As usual, an example helps. Figure 7.8 shows this colour filling command in glorious action. Line 1Ø sets up to Mode 7, giving you a

```
10 GRAPHICS 7
20 SETCOLOR 0,9,12:COLOR 1
30 PLOT 159,20:DRAWTO 159,0
40 DRAWTO 0,0:POSITION 0,20
50 POKE 765,1:XIO 18,#6,0,0,"S:"
60 COLOR 2:PLOT 159,79:DRAWTO 159,20
70 DRAWTO 0,20:POSITION 0,79
80 POKE 765,2:XIO 18,#6,0,0,"S:"
90 A=0.5:X1=60:X2=70:COLOR 3
100 FOR Y=20 TO 79
110 PLOT INT(X1-A),Y:DRAWTO INT(X2+A),
Y
120 A=A+0.5
130 NEXT Y
```

Fig. 7.8. Using XIO for colour filling, and an example of an alternative method.

resolution of 160 × 80, with split screen operation. Line 2Ø then uses SETCOLOR to make register Ø hold colour 9 (light blue) at brightness 12 (almost full brightness). Lines 3Ø and 4Ø follow the XIO instructions, and prepare a rectangular shape. This shape is then filled with colour by line 5Ø, using 1 as the colour number. Remember that the number that is used in this command follows the same method as is used for COLOR – it takes its colour from the register whose number is one less. In this example, then, the colour for the fill is taken from register zero. Lines 6Ø and 7Ø then draw out another rectangular shape, this time using another COLOR number. This is then filled by the XIO instruction in line 8Ø. The last part of the drawing is not so easy to do by using XIO, so I have used it to illustrate another method of colour filling, the old-fashioned method. A loop is set up, and lines drawn in colour from one edge to the other, all the way down the rest of the screen. RUN it, and just watch it go. Impressive, isn't it?

Moving in better circles

Drawing straight lines and boxes is useful, but being able to draw circles greatly extends our artistic range. Atari, has, alas, no CIRCLE

instruction. That's a pity, because a CIRCLE command greatly increases the range of drawings that we can do. As usual in computing, if we don't have a command, we have to create the same effect in another way. The way we have to use is by making a subroutine carry out the operation. The only snag with this method is that the circles are then drawn rather slowly. This is the curse of using BASIC, and it's why so many games programmers go in for machine code, which is difficult but fast. Let's take a look at a couple of subroutines which will draw circles in BASIC.

To start with there are two ways of drawing a circle. One is to plot a set of points that fall round the rim of a circle. Obviously, we would use the PLOT command to do that. The other method is to draw a set of lines around a point. Now if you draw only four lines, the shape is a square, but as you increase the number of lines, the shape starts to look more and more like that of a circle. We can use DRAWTO for this type of circle-drawing subroutine. Whichever method we use, though, you'll have to put up with a bit of trigonometry if you're going to understand it!

After that introduction, take a look at Fig. 7.9. which shows the DRAWTO method in action. Line 1Ø sets up the familiar

```
10 GRAPHICS 7:COLOR 1
20 X=79:Y=39:R=15
30 PLOT X,Y+R:DEG
40 FOR N=0 TO 360
50 DRAWTO X+R*SIN(N),Y+R*COS(N)
60 NEXT N
70 REM TRY DIFFERENT STEPS
```

Fig. 7.9. Drawing a circle with DRAWTO. Try it with STEP 18, and see how much faster it is!

GRAPHICS 7 conditions, and line 2Ø sets the centre of the circle at 79,39, which is the centre of the screen area in this mode (The graphics screen, that is, not counting the text area underneath). The variable R is used for the radius of the circle. That, remember, is the distance from the centre to the outside. Line 3Ø then plots a point on the outside of the circle, and uses DEG to ensure that the Atari will be working with degrees and not radians of angular measurement. What we're going to do now is to draw a set of lines at a constant distance from the centre. We have decided on 360 lines, which is a lot of line drawing! That's what makes the subroutine slow, and if we are using larger pixels, it's usually possible to get away with a smaller number of steps. You still need N = Ø to 36Ø, because there are 360 degrees

round a circle, but you can add STEP 5, or whatever gives acceptable results. The scheme is to find the X and Y numbers of the next point on the rim of the circle. For each angle, the X position is X × R*SIN(N) and the Y number is Y + R*COS(N). N is the angle number, remember, and X and Y are the numbers which give the position of the centre of the circle. The DRAWTO causes a straight line to be drawn from the previous point to this new one on each pass of the loop. Try the subroutine as it is, and then try it with STEP9, and then with STEP18. These different STEP numbers will show you how far you can get from a truly circular shape and still have it look reasonably circular at normal viewing distances. Try the same in other modes, too, but remember that you will have to pick other values of X and Y in other modes.

Figure 7.10 illustrates another way of creating a circular shape, by plotting points. This time, it's algebra rather than trigonometry that

```
10 GRAPHICS 7:COLOR 1
20 X=79:Y=39:R=15
30 PLOT X-R,Y
40 FOR J=(X-R) TO (X+R)
50 P=SQR(R^2-(J-X)^2)
60 PLOT J,Y+P:PLOT J,Y-P
70 NEXT J
80 REM GOOD FOR PLANET SHAPES!
```

Fig. 7.10. Drawing a circle with PLOT. It's slow, but it has its uses.

is needed. So that you can compare it directly with the first method, I have used Mode 7 again. The values of X, Y and R are chosen as before, and a starting point is plotted in line 3ϕ. The loop then takes values of X − R to X + R, which means from one side of the circle to the other. For each value of this X distance, a Y value is calculated in line 5ϕ. The basis of the method is a 2500-year-old bit of geometry called Pythagoras' Theorem. If you've forgotten it, then you can bone up on it. If you never met it before, trust me! That's what has been used in the formula in line 5ϕ. When the Y number has been found in this way, the point can be plotted in line 6ϕ, and then we can move on to the next value along the X direction. It's rather slow, and the circle doesn't look very good because the distance between the dots is not constant. Take your pick!

Chapter 8
Sounds of Music

The ability to produce sound is an essential feature of all modern computers. The sound of the Atari comes from the loudspeaker of the TV receiver that you use to see the display, so you have more control over the volume of this sound than is possible with a lot of other computers.

What we call sound is the result of rapid changes of the pressure of the air round our ears. We don't notice these pressure changes unless they are fairly fast, and we measure the rate in terms of cycles per second, or hertz. A cycle of a wave is a set of changes, first in one direction, then in the other and back to normal, which we can illustrate by the graph in Fig.8.1. The reason that we talk about a sound 'wave' is because the shape of this graph is a wave shape.

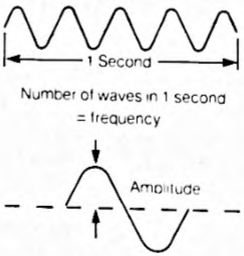


Fig. 8.1. Sound waveforms, showing how the air pressure changes with time. The number of changes per second is called the *frequency*. The amount of the change is the *amplitude*.

The *frequency* of sound is its number of hertz – the number of cycles of changing air pressure per second. If this amount is less than about 20 Hertz, we simply can't hear it, though it can still have disturbing effects. We can hear the effect of pressure waves in the air at frequencies above 20 hertz, going up to about 15000 hertz. The

frequency of the waves corresponds to what we sense as the 'pitch' of a note. A low frequency of 80 to 120 hertz corresponds to a low-pitch bass note. A frequency of 400 or above corresponds to a high-pitch treble note.

The amount of pressure change determines what we call the loudness of a note. This is measured in terms of the *amplitude*, which is the maximum change of pressure of the air from its normal value. For complete control over the generation of sound, we need to be able to specify the amplitude, frequency, shape of wave, and also the way that the amplitude of the note changes during the time when it sounds.

The Atari has one sound instruction, which reasonably enough uses the command word SOUND. This instruction word has to be followed by four numbers, separated from each other by commas. The first number is a *channel number*. The Atari can produce up to four different notes at the same time, and it does this by allocating four different channels of sound. When you specify a sound on a channel, it replaces any note that was previously playing on that channel. The allowable channel numbers are Ø to 3, and if you try to use larger numbers you will get an error message.

The second number is the *pitch number*. This can take any value in the range Ø to 255. Using Ø produces the highest of the normal range of notes, 255 gives the lowest note. Your Atari BASIC manual shows (page 5) how these numbers are related to the piano scale. You will need to use this diagram if you want to create music on your Atari.

The third number is called the *distortion number*. It controls the *type* of note that you hear. A value of 1Ø for this number produces the familiar 'electronic organ' type of note, but by changing the value of this number, you can obtain other types of sound, some very useful for sound effects. The range of numbers that you can use is Ø to 14, and only *even* numbers can be used usefully. Using an odd number will produce only clicks.

The last number, the *volume number*, controls relative volume. The 'absolute volume' of the sound that you hear is fixed by the setting of the volume control of your TV receiver. The volume number in the SOUND instruction sets 'relative volume'. You find that in music some notes are soft, some loud. If you are going to reproduce this effect with your computer, you need a way of controlling volume by the SOUND command. It's not really good enough to have to flash a message on the screen that says 'Turn

volume up, loud bit coming'. If you think that sounds funny, you maybe don't know that there are some computers that don't allow any form of control of volume! The range of volume number is Ø to 15. Zero gives silence, 15 gives full volume, but if you are playing more than one note at a time, then the total of all the volume numbers should not be more than 32. The reason is that the circuits for the SOUND channels can be overloaded if you use a total of more than this amount.

Take a note, any note ...

Let's start our investigation of the SOUND instruction with a rising pitch of note which makes a useful warning, or a 'something about to happen' note. This is illustrated in Fig. 8.2. The loop that starts in line

```
10 FOR J=255 TO 1 STEP -1
20 SOUND 0,J,10,8
30 FOR Z=1 TO 5:NEXT Z
40 NEXT J
50 SOUND 0,0,0,0
```

Fig. 8.2. A warning note, of increasing pitch.

1Ø uses values of J that range from 255 to Ø, the full range that the SOUND instruction permits. These are the numbers that we shall use as pitch numbers in the SOUND instruction in line 2Ø. Along with this changing pitch, we are using Channel Ø, a distortion value of 1Ø (pure note), and a volume number of 8. A delay loop in line 3Ø determines for how long the sound will continue. This is important, because when you start a note playing, the computer hands over control to other circuits, and the note continues playing until you stop it by another command. Pressing BREAK will not stop a sound if you haven't used a statement like SOUND Ø,Ø,Ø,Ø to stop it. In such a case, only the END statement, or using NEW or RUN will do so.

Figure 8.3 shows a program that produces a warbling note. This is

```
10 FOR J=1 TO 200
20 SOUND 0,121,10,8
30 FOR Z=1 TO 20:NEXT Z
40 SOUND 0,130,10,8
50 NEXT J
60 SOUND 0,0,0,0
```

Fig. 8.3. A warbling note program.

particularly useful for attracting attention, or for announcing a score in a game. For some reason, a warbling note attracts our attention more than a single note, which is why a warbling note was chosen for the later types of telephones. The warble in this program uses the loop that starts in line 1Ø. This sounds 2ØØ notes, which are short with a duration fixed by the loop in line 3Ø. The two pitch numbers that have been chosen in this example are 121 and 13Ø.

The special effects

Producing music, particularly in four-part harmony, is a rather specialised topic unless you have some musical training. For that reason, I shall not go into details of producing music in this book. Instead, I'll concentrate on sound effects, which are normally what the sound channels are most used for. Sound effects on the Atari are easy to create. What is difficult is getting the sound effect that you want! The trouble is that it's much more difficult to describe a sound than it is to describe a picture. If you want to make use of sound effects, then, you have to get your ears trained to what can be produced by different SOUND instructions. In particular, you need to listen to the effects that are produced when different values of the distortion number are used.

Figure 8.4 plays one single note, with the same values of pitch number, channel and volume throughout. At each pass through the

```
10 ? "}":FOR J=0 TO 14 STEP 2
20 POSITION 5,5:? "Distortion ";J
30 SOUND 0,121,J,8
40 FOR Z=1 TO 1000:NEXT Z
50 NEXT J
60 SOUND 0,0,0,0
```

Fig. 8.4. The effect of the distortion number on one note. Try the effect on other notes – it's not the same for all notes, and the results can be very interesting.

loop, however, the distortion number is changed. These numbers are printed on the screen so that you can see which number produces which effect. You'll find that some distortion numbers have the effect of making the note sound at a different pitch. This effect is complicated, and all you can do is to investigate it for yourself. Try a range of different pitch values with the program of Fig. 8.4, and listen as the different distortion numbers are used. Make a note of any effects that you particularly like. It's only in this way that you will gradually be able to associate different combinations of pitch number

and distortion number with the sounds that they produce. Distortion number 12 is particularly effective in this way. When it is used, you can produce some very low notes, much lower than you can get by using a pitch number of 255 with a pure tone (distortion 1Ø).

Some useful effects

The sound effects that will be most useful to you are probably ones that you find by accident. The range of sound effects that the Atari can produce is so vast that no-one could claim to be able to produce all the possible sounds. It helps a lot, however, if you know how to produce some of the 'standards', the sound effects that are widely used. You can use these as the basis for experiment, varying the pitch and distortion numbers to see if the sound can be made into something that you want. Do you need the sound of an ion-gun being fired on Venus? The sound of a drowning mammoth? An orchestra playing in compressed helium? With these starter-sounds, you can work your way to whatever it is you want.

To start with, try Fig.8.5, which produces the sound of waves breaking on a shore. It's excellent for desert islands as it is, and you

```
10 FOR Z=1 TO 5:FOR N=50 TO 0 STEP -1
20 SOUND 0,N,8,10
30 FOR J=1 TO 50:NEXT J
40 NEXT N:NEXT Z
50 SOUND 0,0,0,0
```

Fig. 8.5. Surf on the shore, just the thing for desert islands.

can get a variety of other wave noises by changing numbers. In the example, the number of waves that you hear is controlled by variable Z. The pitch of the sound is changed as it plays, which is what causes the breaking effect. This pitch number is controlled by variable N. A distortion number of 8 selects the sort of hissing sound that we need (a form of 'white noise'), and the delay loop in line 3Ø provides the right amount of time for the wave. Line 5Ø stops the sounds. There are a lot of numbers that can be changed in this one, so you have plenty of scope for experimenting.

Riding further West, how about some gunshot sounds? A gunshot is a short sound with a large volume, and Fig. 8.6 illustrates how it can be produced. This program produces six shots (what else, Marshall?) by using the loop that starts in 1Ø. The sound uses maximum volume, a pitch number of 1, and the distortion number

```
10 FOR X=1 TO 6
20 SOUND 0,1,0,15
30 FOR J=1 TO 80:NEXT J
40 SOUND 0,0,0,0
50 FOR J=1 TO 500:NEXT J
60 NEXT X
70 END
```

Fig. 8.6. Gunshots for making your Corral O.K.

zero. Different pitch numbers in this sound have surprisingly little effect, but it's worth experimenting with to produce the type of gunshot that you want. Distortion numbers Ø and 2 are particularly useful for this type of 'bang, bang' sound.

How about a visit to Brands Hatch? A good engine-revving sound is always a useful one, and the program of Fig. 8.7 will provide you with lots of ideas. It uses full volume (of course), and a pitch number

```
10 FOR D=1 TO 500
20 SOUND 0,20,0,15
30 SOUND 0,20,1,15
40 NEXT D
50 SOUND 0,0,0,0
```

Fig. 8.7. Engine noises for racing cars!

of 2Ø, but with the distortion number being changed between Ø and 1. Try it – and hear the sound of the starting grid. Better still, add more of the same sound on other channels with rather different pitch numbers. It's a great accompaniment to those motor-racing games!

Final note

Sound is different. As I said, it's easy to tell you what colour a number can produce, or what shape will be given by a set of DRAWTO instructions. Nothing, however, can convey to you on paper what something sounds like. Apart from anything else, a lot depends on the size of the loudspeaker in your TV receiver. Only a large loudspeaker will do justice to the very low notes that can be produced by juggling with different pitch and distortion numbers. Your own ears are also different from mine, and from the ears of anyone else. Getting a sound that satisfies you, then, requires a lot of work. You need to take some starting point, and note the effect of changing the SOUND control numbers. I mean *note*, too. If you don't write down on paper what each particular change does, you'll soon forget. When this

happens, you can waste a lot of time repeating experiments that you have tried earlier. If you make a note of the effect of a change, you can decide whether it goes in the direction that you want or not. If it does, then try some more. If it doesn't, then it's likely that you'll want that effect some other time. Keeping a note ensures that you will be able to get back to the effect. It's even more useful if you can make a cassette recording of each SOUND program that you use. This way, you can reproduce both the sound and the program that produces it each time you want to experiment.

Chapter 9
Do It Yourself

You can get a lot of enjoyment from your Atari when you use it to enter programs from cassettes or cartridges that you have bought. You can obtain even more enjoyment from typing in programs that you have seen printed in magazines. Even more rewarding is modifying one of these programs so that it behaves in a rather different way, making it do what suits you. The pinnacle of satisfaction, as far as computing is concerned, however, is achieved when you design your own programs. These don't have to be masterpieces. Just to have decided what you want, written it as a program, entered it and made it work is enough. It's 100% your own work, and you'll enjoy it all the more for that.

Now I can't tell in advance what your interests in programs might be. Some readers might want to design programs that will keep tabs on a stamp collection, a record collection, a set of notes on food preparation or the technical details of vintage steam locomotives. Programs of this type are called *database* programs, because they need a lot of data items to be typed in and recorded. On the other hand, you might be interested in games, colour patterns, drawings, sound, or other programs that require shapes to move around the screen. Programs of that type need a lot of experience in the use of graphics and sound. What we are going to look at in this section is the database type of program, because it's designed in a way that can be used for all other types of programs.

Two points are important here. One is that experience counts in this design business. If you make your first efforts at design as simple as possible, you'll learn much more from them. That's because you're more likely to succeed with a simple program first time round. You'll learn more from designing a simple program that works than from an elaborate program that never seems to do what it should. The second

point is that program design has to start with the computer switched off, preferably in another room! The reason is that program design needs planning, and you can't plan properly when you have temptation in the shape of a keyboard in front of you. Get away from it!

Put it on paper

We start, then, with a pad of paper. For myself, I use a 'student's pad' which is punched so that I can put sheets into a file. This way I can keep the sheets tidy, and add to them as I need. I can also throw away any sheets I don't need, which is just as important. Yes, I said sheets! Even a very simple program is probably going to need more than one sheet of paper for its design. If you then go in for more elaborate programs, you may easily find yourself with a couple of dozen sheets of planning and of listing before you get to the keyboard. Just to make the exercise more interesting, I'll take an example of a program, and design it as we go. This will be a fairly simple program, but it will illustrate all the skills that you need. I'll follow it up with a listing of another relatively simple program of a different kind, with less explanation. You then have two 'frameworks' around which you can build programs for yourself.

Start, then, by writing down what you expect the program to do. You might think that you don't need to do this, because you know what you want, but you'd be surprised. There's an old saying about not being able to see the wood for the trees, and it applies very forcefully to designing programs. If you don't write down what you expect a program to do, it's odds on the program will never do it! The reason is that you get so involved in details when you start writing the lines of BASIC that it's astonishingly easy to forget what it's all for. If you write it down, you'll have a goal to aim for, and that's as important in program design as it is in life. Don't just dash down a few words. Take some time about it, and consider what you want the program to be able to do. If you don't know, you can't program it!

As an example, take a look at Fig. 9.1. This shows a program outline plan for a simple game. The aim of the game is to become familiar with units of money around the world. The program plan shows what I expect of this game. It must present the name of a country on the screen, and then ask what the main unit of money is (like pound, dollar, mark and so on). A little bit more thought produces some additional points. The name of the currency will have to be correctly spelled. A little bit of trickery will be needed to prevent

Aims:

- Present the name of a country on the screen
- Ask what the main unit of currency is
- Must be correctly spelled
- Must avoid user being able to read answer from listing
- One point for each correct answer
- One additional try allowed
- Keep track of attempts
- Present score as number of successes out of number of attempts
- Pick country names at random

Fig. 9.1. A program outline plan. This is your starter!

the user (son, daughter, brother, sister) from finding the answers by typing LIST and looking for the DATA lines. Every game must have some sort of scoring system, so we allow one point for each correct answer. Since spelling is important, perhaps we should allow more than one try at each question. Finally, we should keep track of the number of attempts and the number of correct answers, and present this as the score at the end of each game. Now this is about as much detail as we need, unless we want to make the game more elaborate. For a first effort, this is quite enough. How do we start the design from this point on?

The answer is to design the program the way an artist paints a picture. That means designing the outlines first, and the details later. The outlines of this program are the steps that make up the sequence of actions. We shall, for example, want to have a title displayed. Give the user time to read this, and then show instructions. There's little doubt that we shall want to do things like assign variable names,

- Display title, then instructions
- Present name of country on screen
- Ask for unit of currency
- Input for reply
- Compare with correct answer
- If correct, ask if another one is wanted
- If incorrect, give one more try
- If second try is incorrect, select another country
- Ends when user types N in reply to 'Do you want another one?'

Fig. 9.2. The next stage in expanding the outline.

dimension arrays, and other such preparation. We then need to play the game. The next thing is to find the score, and then ask the user if another game is wanted. Yes, you have to put it all down on paper! Figure 9.2 shows what this might look like at this stage.

The BASIC foundations

Now, at last, we can start writing a chunk of program. This will just be a foundation, though. What you must avoid at all costs is filling pages with BASIC lines at this stage. As any builder will tell you, the foundation counts for a lot. Get it right, and you have decided how good the rest of the structure will be. The main thing you have to avoid now is building a wall before the foundation is complete!

Figure 9.3 shows what you should aim for at this stage. There are

```
10 ? "}":GOSUB 1000
11 REM TITLE
15 OPEN #1,4,0,"K:"
20 GOSUB 1200
21 REM INSTRUCTIONS
30 GOSUB 1400
31 REM SETUP
40 GOSUB 2000
41 REM PLAY
50 GOSUB 3000
51 REM SCORE
60 GOSUB 4000
61 REM ANOTHER?
70 IF K=89 THEN 40
100 END
```

Fig. 9.3. A 'core' or 'foundation' program for the example.

only fifteen lines of program here, and that's as much as you want. This is a foundation, remember! It's also a program that is being developed, so we've hung some 'danger – men at work' signs around. These take the form of the lines that start with REM. REM means REMinder, and any line of a program that starts with REM will be ignored by the computer. This means that you can type whatever you like following REM, and the point of it all is to allow you to put notes in with the program. These notes will not be printed on the screen when you are using the program, and you will see them only when you LIST. In Fig. 9.3, I have put the REM notes on lines which are numbered just 1 more than the main lines. This way, I can remove all the REM lines later. How much later? When the program is complete, tested, and working perfectly. REMs are useful, but they

make a program take up more space in memory, and run slightly slower. I always like to keep one copy of a program with the REMs in place, and another 'working' copy which has no REMs. That way I have a fast and efficient program for everyday use, and a full-detailed version that I can use if I want to make changes.

Let's get back to the program itself. As you can see, it consists of a set of GOSUB instructions, with references to lines that we haven't written yet. That's intentional. What we want at this point, remember, is foundations. The program follows the plan of Fig. 9.2 exactly, and the only part that is not committed to a GOSUB is the IF in line 7Ø. What we shall do is to write a subroutine which will use GET to look for a 'Y' or 'N' being pressed, and line 7Ø deals with the answer. What's the question? Why, it's the 'Do you want another game' step that we planned for earlier.

Take a good long look at this piece of program, because it's important. The use of all the subroutines means that we can check this program easily – there isn't much to go wrong with it. We can now decide in what order we are going to write the subroutines. The wrong order, in practically every example, is the order in which they appear. Always write the title and instructions *last*, because they are the least important to you at this stage. In any case, if you write them too early, it's odds on that you will have some bright ideas about improving the game soon enough, and you will have to write the instructions all over again. A good idea at this stage is to write a line such as:

```
9 GOTO 3Ø
```

which will cause the program to skip over the title and instructions. This saves a lot of time when you are testing the program, because you don't have the delay of printing the title and instructions each time you run it.

The next step is to get to the keyboard (at last, at last!) and enter this core program. If you use the GOTO step to skip round the title and instructions temporarily, you can then put in simple PRINT lines at each subroutine line number. We did this, you remember, in the program of Fig. 4.14, so you know how to go about it. This allows you to test your core program and be sure that it will work before you go any further.

The next step is to record this core program. You can then build up the rest of the program by adding to the core. If you have the core

recorded, then you can load this into your Atari, type in one of the subroutines, and then test. When you are satisfied that it works, you can record the whole lot on another cassette. Next time you want to add a subroutine, you start with this version, and so on. This way, you keep tapes of a steadily growing program, with each stage tested and known to work. If there is a thunderstorm during these operations, causing a power failure, then all you can lose is the new material that has not yet been recorded. That's a lot better than losing the lot!

Subroutine routine

The next thing we have to do is to design the subroutines. Now some of these may not need much designing. Take, for example, the subroutine that is to be placed in line 4000. This is just our familiar GET routine, along with a bit of PRINT, so we can deal with it right away. Type it in (Fig.9.4), and now test the core program with this subroutine in place.

```
4000 ? "Would you like another one?":?
 "Please answer Y or N."
4010 GET #1,K:RETURN
```

Fig. 9.4. The subroutine for line 4000.

Now we come to what you might think is the hardest part of the job – the subroutine which carries out the Play action. In fact, you don't have to learn anything new to do this. The Play subroutine is designed in exactly the same way as we designed the core program. That means we have to write down what we expect it to do, and then arrange the steps that will carry out the action. If there's anything that seems to need more thought, we can relegate it to a subroutine to be dealt with later.

As an example, take a look at Fig. 9.5. This is a plan for the Play subroutine, which also includes information that we shall need for the

- Keep answers as a set of ASCII codes in an array, NUM$().
- Keep list of countries in another array, Q$().
- The number which selects the country will also select the answer.
- Use variable TRY for recording tries, and SCORE to keep score.
- Use variable GO to record the number of attempts at one question.

Fig. 9.5. Planning the 'Play' subroutine.

setting-up steps. The first item is the result of a bit of thought. We wanted, you remember, to be sure that some smart user would not cheat by looking up the answers in the DATA lines. The simplest deterrent is to make the answers in the form of ASCII codes. It won't deter the more skilled, but it will do for starters.

The next one is that we shall keep the names of the countries, and the ASCII codes for the currencies, in two long strings. This has several advantages. One of them is that it's beautifully easy to select one at random if we do this. The other is that it also makes it easy to match the answers to the questions. If we use the number variable V for the number of an item, then we can use an array item, ARRAY (V) to find the name that corresponds to it. How do we do this? Well, if each number in the array is the starting position of the first letter of the country, then we can use two numbers to find the country. Confused? Well, suppose the country is ANTIGUA, and that the first letter A is character number 35 in the long string. The first letter of the next country will be character number 42. If we slice the string between 35 and 41, then, we will get the name of the country. We would have the number 35 as one item in a number array, and 42 as the next. What we need is to slice between item V and (item V+1) — 1. Item V is 35, and item V+1 is 42. We can use exactly the same scheme for the answers!

The next thing that the plan settles is the names that we shall use for variables. It always helps if we can use names that remind us of what the variables are supposed to represent. In this case, using SCORE for the score and TRY for the number of tries looks self-explanatory. The third one, 'GO' is one that we shall use to count how many times one question is attempted. Finally, we decide on names for the strings that will hold the country names and the currency codes - Q$ and NUM$.

Play it again, Sam

Figure 9.6 shows what I've ended up with as a result of the plan in Fig. 9.5. The steps are to pick a random number, use it to print a country name, and then find the answer. That's all, because the checking of the answer and the scoring is dealt with by another subroutine. Always try to split up the program as much as possible, so that you don't have to write huge chunks at a time. As it is, I've had to put another subroutine into this one to keep things short.

```
2000 GO=0:V=1+INT(10*RND(0))
2001 REM PICK AT RANDOM
2010 ? "}":POSITION 6,5:? "The country
 is_ ";
2020 ? Q$(ARRAY(V),ARRAY(V+1)-1)
2030 POSITION 8,8:? "The currency is_
";
2040 INPUT CUR$:TRY=TRY+1
2050 GOSUB 5000
2051 REM FIND CORRECT ANSWER
2060 RETURN
```

Fig. 9.6. The program lines for the 'Play' subroutine.

We start in line 2ØØØ by picking a number at random, but lying between 1 and 1Ø. As before, we use line 2ØØ1 to hold a REM that reminds us of what's going on. Line 2Ø1Ø does a bit of printing, and then line 2Ø2Ø picks the country name out of Q$. We print the name of the country that corresponds to the random number, and ask for an answer, the currency of that country. The last section of line 2Ø4Ø counts the number of attempts. This is the logical place to put this step, because we want to increment the count each time there is an answer. Now it's chicken-out time. I don't want to get involved in the reading of ASCII codes right now, so I'll leave it to a subroutine, starting in line 5ØØØ, which I'll write later. The REM in line 2Ø51 reminds me what this new subroutine will have to do, and the Play subroutine ends with the usual RETURN.

Down among the details

With the Play subroutine safely on tape, we can think now about the details. The first one to look at should be one that precedes or follows the Play step, and I've chosen the Score routine. As usual, it has to be planned, and Fig. 9.7 shows the plan. Each time that there is a correct

1. For correct answer, increment SCORE.
2. For first incorrect answer, with GO=0. allow another try and make GO=1.
3. For second incorrect answer, when GO=1, pass the next question and make GO=0 again.

Fig. 9.7. Planning the 'Score' subroutine.

answer, the number variable 'SCORE' will be incremented, and we can go back to the main program. More is needed if the answer does

not match exactly. We need to print a message, and allow another go. If the result of this next go is not correct, that's an end to the attempts.

Figure 9.8 shows the program subroutine developed from this plan. Line 3ØØØ deals with a correct answer by comparing your

```
3000 IF ANS$=CUR$ THEN SCORE=SCORE+1:?
 :? M1$;" ";SCORE:? "IN ";TRY;" ATTEMP
TS.":GOSUB 7000:GOTO 3030
3010 IF GO=0 THEN ? M2$:GOSUB 7000:GO=
1:GOSUB 2010:GOTO 3000
3020 GO=0:? "No luck, try the next one
."
3030 RETURN
```

Fig. 9.8. The 'Score' subroutine written.

answer CUR$ with the correct answer, ANS$. The GOTO 3Ø3Ø ensures that if the answer was correct, the rest of the subroutine is skipped, and the subroutine returns. If the answer is not correct, though, line 3Ø1Ø swings into action. This prints a message, and then calls the subroutine at line 2Ø1Ø again so that the user can make another answer entry. The GOTO 3ØØØ at the end of line 3Ø1Ø then tests this answer again.

Now there's a piece of cunning here. The number variable 'GO' must start with a value of Ø (make a note of it!). When there is an incorrect answer, however, and 'GO' is still Ø, line 3Ø1Ø is carried out. One of the actions of line 3Ø1Ø, however, is to 'GO' to 1. When you answer again, with GO = 1, line 3ØØØ will be used, and if your second answer is wrong, line 3Ø1Ø cannot be used, because 'GO' is not zero. The next line that is tried, then, is 3Ø2Ø. This puts 'GO' back to zero for the next round, prints a sympathetic message, pauses, and then lets the subroutine return in line 3Ø3Ø.

Now that we've got the bit between our teeth, we can polish off the rest of the subroutines. Figure 9.9 shows the subroutine that deals with dimensioning and arrays. Line 14ØØ sets all the variables for the scoring system to zero. Lines 14Ø5 and 14Ø6 dimension all the strings and arrays that are used for the names. Lines 141Ø and 1412 create strings that will be used for printing messages. Lines 1415 to 1425 then create the strings that hold the names of the countries and the ASCII codes for their currencies, using the same order. The codes have to be put into two strings, and then combined. This is because you are limited to typing about 120 characters following one line number. The combining operation is done in line 143Ø. Finally, the array of numbers that holds the positions of question and answer

```
1400 TRY=0:SCORE=0:GO=0
1405 DIM M2$(64),M1$(32),Q$(80),NUM$(1
60),NUN$(70),ARRAY(45)
1406 DIM CUR$(20),ANS$(20),REP$(40)
1410 M2$="Not correct_ but it might be
 your      spelling. Try again- free!"
1412 M1$="Correct- your score is now "
1415 Q$="AUSTRIABOLIVIACHILECOSTA RICA
DENMARKHAITIIRAQMOROCCONICARAGUAPOLAND
"
1420 NUM$="836772737676737871806983796
983678568796779767978758279786 9"
1425 NUN$="71798582686968737865826873 8
272657767798268796665907679848 9"
1430 NUM$(LEN(NUM$)+1)=NUN$
1435 FOR N=1 TO 22:READ A:ARRAY(N)=A:N
EXT N
1450 RETURN
```

Fig. 9.9. The dimensioning and array subroutine.

words is read in line 1435, and the subroutine returns in line 145ø.

Next comes the business of finding the answer. We have planned this, so it shouldn't need too much hassle. Figure 9.10 shows the program lines. The variable V is the one that we have selected at

```
5000 Q=V+11:REP$=NUM$(ARRAY(Q),ARRAY(Q
+1)-1)
5010 ANS$="":FOR N=1 TO LEN(REP$) STEP
 2
5015 J=(N+1)/2
5020 ANS$(J,J)=CHR$(VAL(REP$(N,N+1)))
5030 NEXT N
5040 RETURN
```

Fig. 9.10. Checking the answer.

random, and we add 11 to it so as to find the corresponding answer number. Why eleven? Since there are ten questions, the eleventh item is the answer to the first question, hence the eleven. We extract the string of numers from NUM$, and assign it to REP$, which is the reply for this question. We now have to convert each number in REP$ into a letter, and add each letter to the string ANS$, which starts off blank in line 5ø1ø. Lines 5ø1ø to 5ø3ø build up the answer string. There are two digits in each ASCII code, which is why we use STEP2 in the loop. The limit of the loop is the number of characters in REP$, and we obtain this by using LEN. Line 5ø15 is a formula for J which picks out values of 1,2,3 ... and so on as N goes in steps of 1,3,5 ... etc. Line 5ø2ø then places each letter into the answer string,

using J as the position number. The letter has to be found by extracting the number from REP$, using REP$ (N,N+1), and then using VAL to get the number form, then CHR$ to convert to a letter. That's the hard work over!

Figure 9.11 is the subroutine for the instructions, and Fig. 9.12 is

```
1200 ? "}":? "                    INSTRUCT
IONS"
1210 ? :? "You will be given the name
of a ":? "country. You are asked to ty
pe the"
1220 ? "name of its currency (use CAPI
TALS)":? "and then press RETURN. You a
re"
1230 ? "allowed two attempts at each c
ountry.":? "The computer will keep sco
re."
1240 ? "Press any letter key to start.
"
1250 GET #1,X:RETURN
```

Fig. 9.11. The instructions – always leave these until you have almost finished.

```
1000 ? "}"
1010 POSITION 14,5:? "CURRENCY"
1020 FOR N=1 TO 2000:NEXT N
1040 RETURN
```

Fig. 9.12. The title subroutine.

the title subroutine. Each of them include a pause. Finally, Fig. 9.13 shows the DATA lines and the subroutine which is used to provide a delay after each answer. Now we can put it all together, and try it out.

```
6000 DATA 1,8,15,20,30,37,42,46,53,62,
68,1,19,27,39,49,59,71,81,93,107,117
7000 FOR N=1 TO 400:NEXT N:RETURN
```

Fig. 9.13. The DATA lines that are needed, along with a time delay subroutine.

Because it's been designed in sections like this, it's easy for you to modify it. You can use different DATA, for example. You can use a lot more data – but remember to change the DIM statements.

You can, of course, make this a question-and-answer game on something entirely different, just by changing the data and the instructions. Take this as a sort of BASIC 'Meccano set' to reconstruct any way you like. It will give you some idea of the sense of achievement that you can get from mastering your Atari!

A file program

The following program, *Address Book*, is of the data filing type. Although it is fairly long, it consists of sections that you can enter one by one. This also allows you to change the program to suit yourself. It allows you to enter names, addresses and telephone numbers for your friends, and to record all of this data on tape. The technique is the one that we looked at earlier, using fixed amounts of characters for each entry. One a 16K machine, you can use a reasonable number of entries (about 100, if you dimension accordingly). You can play back the data tape, and use this to find a name, address and phone number, even if you can only remember the first letter of the surname!

The program follows conventional lines, but the displays look a lot more interesting than those in our previous example. One point which may puzzle you, however, is the instruction POP in line 122Ø. The subroutine in line 12ØØ causes a time delay, which can be cut short by pressing the spacebar. The time delay is generated by a FOR ... NEXT loop, and pressing the spacebar causes the loop to be abandoned while the counting is in progress. This can cause problems unless soemthing is done to 'tidy up' the remains of the count numbers in the memory. POP does this, ensuring that all will be well next time a loop is used.

The most important thing of all, though, is that this program has been designed using the same principles as we dealt with in detail in this chapter. See if you can trace out the design steps for yourself. Would you like to modify the program? Could you change it so that it could be used to catalogue a stamp collection, for example? Better still, could you design a program that does what *you* want? That's the real fun of computing!

Address Book

```
10 DIM CHUM$(3900),N$(50)
20 DIM FIND$(20),Y$(1)
30 DIM IN$(85)
100 ? "}":POKE 82,0:OPEN #1,4,0,"K:":P
OKE 752,1
110 GOSUB 1000:REM TITLE
120 GOSUB 1100:REM INSTRUCTIONS
130 GOSUB 1300:REM MENU
140 ON K-48 GOSUB 2000,3000
150 POKE 752,0:END
1000 GRAPHICS 2:POKE 756,224:FOR N=1 T
O 11:READ D
```

```
1010 COLOR D:PLOT 3+N,5
1020 NEXT N:? "Press spacebar to conti
nue, or just wait"
1030 GOSUB 1200
1040 RETURN
1050 DATA 65,68,68,114,101,115,211,194
,207,239,235
1100 GRAPHICS 1:SETCOLOR 4,5,4:SETCOLO
R 0,1,12
1110 POSITION 4,0:PRINT #6;"INSTRUCTIO
NS"
1120 POSITION 1,2:PRINT #6;"Choose a m
enu item."
1130 PRINT #6;"You can write names":PR
INT #6;"and addresses and"
1140 PRINT #6;"record them. You can":P
RINT #6;"also read back data"
1150 PRINT #6;"as you need it."
1160 ? "Press spacebar to":? "continue
, or just wait."
1170 GOSUB 1200
1180 RETURN
1200 FOR J=1 TO 5000
1210 IF PEEK(764)=255 THEN NEXT J
1220 POP :POKE 764,255
1230 RETURN
1300 GRAPHICS 2:SETCOLOR 4,11,2:SETCOL
OR 0,7,12
1310 POSITION 7,1:PRINT #6;"MENU"
1320 POSITION 1,3:PRINT #6;"1. MAKE LI
ST."
1330 POSITION 1,5:PRINT #6;"2. USE LIS
T."
1340 POSITION 1,7:PRINT #6;"CHOOSE BY
NUMBER"
1350 ? "DON'T USE RETURN!"
1360 GET #1,K
1370 IF K<49 OR K>50 THEN ? "WRONG VAL
UE!. CHOOSE 1 OR 2 ONLY.":? "PLEASE TR
Y AGAIN.":GOTO 1340
1380 RETURN
2000 GRAPHICS 0:SETCOLOR 2,6,4:SETCOLO
R 4,6,4:SETCOLOR 1,6,10
2010 POSITION 1,1:? "Please enter name
s, addresses and ":? "telephone number
s as instructed in":? "the book."
2020 ? "Type END to stop entry."
2030 REM YOU CAN ADD MORE!
2050 POSITION 1,22:? "Press spacebar t
o continue, or wait!"
2060 GOSUB 1200
2070 COUNT=0
2080 GOSUB 2200
```

```
2090 ? "Surname,forename,please.":MAX=
20:GOSUB 7000:IF N$="END" THEN 2170
2100 GOSUB 5000
2110 GOSUB 6000:GOSUB 2200
2120 ? "Address-use commas to separate
 lines":MAX=50:GOSUB 7000:GOSUB 5000
2130 GOSUB 6000:GOSUB 2200
2140 ? "Phone number, please.":MAX=15:
GOSUB 7000:GOSUB 5000
2150 GOSUB 6000
2160 COUNT=COUNT+1:? "You have made ";
COUNT;" entries.":GOSUB 1200:IF COUNT<
50 THEN 2080
2170 GOSUB 8000
2180 RETURN
2200 ? "}":POSITION 1,3:RETURN
3000 GRAPHICS 0:POSITION 2,1
3010 ? "First you need to replay the d
ata."
3020 ? "Place the rewound data cassett
e in place"
3030 ? "Press PLAY, then RETURN."
3040 REM MORE IF NEEDED
3050 GOSUB 9000
3080 ? "Now we're ready _"
3090 ? "Press spacebar to go."
3100 GOSUB 1200:? "}":POSITION 1,2
3110 ? "What name would you like?"
3120 ? "Remember to type it accurately
 - but ":? "you needn't type it all!"
3130 ? "Type END to finish."
3140 INPUT FIND$:L=LEN(FIND$):IF FIND$
="END" THEN 3210
3150 FOR N=1 TO 85*COUNT STEP 85
3160 IF CHUM$(N,N+L-1)=FIND$ THEN 3500
3170 NEXT N
3180 ? "No such name in this file."
3190 ? "Please try again _"
3200 GOTO 3100
3210 RETURN
3500 FOR X=0 TO 84:Y$=CHUM$(N+X,N+X)
3510 ? Y$;
3520 IF Y$="," THEN PRINT
3530 IF X=19 OR X=69 THEN PRINT
3540 NEXT X
3550 ? :? "Press spacebar to continue.
..":GOSUB 1200:GOTO 3110
5000 L=LEN(N$)
5010 IF L=MAX THEN 5040
5020 FOR X=L+1 TO MAX
5030 N$(X,X)=" ":NEXT X
5040 RETURN
6000 CHUM$(LEN(CHUM$)+1)=N$
```

```
6010 RETURN
7000 N=1:N$=""
7010 GET #1,X:IF X=155 THEN 7070
7020 N$(N,N)=CHR$(X)
7030 POSITION 1,20:? "You have ";MAX-N
;" characters left    ":? CHR$(156)
7040 POSITION 1,6:? N$
7050 N=N+1:IF N=MAX THEN 7070
7060 GOTO 7010
7070 RETURN
8000 GOSUB 2200
8010 ? "Please prepare cassette for re
cording"
8020 OPEN #2,8,0,"C:"
8030 PRINT #2;COUNT
8040 FOR N=1 TO 85*COUNT STEP 85
8050 PRINT #2;CHUM$(N,84+N)
8060 NEXT N:CLOSE #2
8070 ? "Now rewind cassette and remove
."
8080 ? "Press spacebar to continue"
8090 GOSUB 1200
8100 RETURN
9000 OPEN #2,4,0,"C:"
9010 INPUT #2;COUNT:CHUM$=""
9020 FOR N=1 TO 85*COUNT STEP 85
9030 INPUT #2;IN$:CHUM$(N,84+N)=IN$
9040 NEXT N
9050 CLOSE #2
9060 RETURN
```

Appendices

Appendix A: Editing

If you make a mistake while you are typing a line of BASIC, you can use the BACK SPACE key to 'rub out' characters. This key, like all the others, repeat its action if you hold it down. You can't repair a mistake in this way, however, if you have entered the line by pressing RETURN. It's also rather frustrating, when you're entering a line, to have to delete most of it just because of a minor error at the start. The editing procedure of the Atari makes it easy to mend mistakes.

(1) Place the line that you want to change on the screen. If you are still working on it, or if you have just pressed RETURN, of course, it will be there already. If not, you can use LIST1ØØ (or whatever the line number is) to get the line on to the screen.

(2) You now have to position the cursor so as to deal with the mistakes. Hold down the CONTROL key. This allows you to use four keys on the right-hand side of the keyboard to move the cursor. These are the +,−,* and = keys, and if you look closely at them, you will see arrows marked in the top left-hand corner of each key. The arrow shows the direction in which the cursor will move when the key is pressed. As usual, the action repeats while the key is held down. Don't let go of the CONTROL KEY! The cursor will 'wrap around'. That means if it goes off the right-hand side of the screen, it will reappear on the left. If it goes off the bottom of the screen, it will reappear at the top.

(3) You can now make repairs. If you have PRIBT, for example, place the cursor over the B, release the CONTROL key, and type N. The N will replace the B on the screen. The change is *not permanent*, however, until you press RETURN (with CONTROL released). If

you have a PRINNT, you will want to delete an N. Place the cursor on the N, keep CONTROL pressed, and tap the DELETE/BACK-SPACE key, then RETURN. If you have a PRNT, you need to insert an I. Place the cursor over the N, and tap the INSERT key. Release CONTROL, and type I, then RETURN. You must release the CONTROL key when you type a letter.

(4) These actions deal with most of the editing that you need. Remember that you can hold down the CONTROL and INSERT keys to create a larger space, so as to insert a complete new command, not just a single letter. If you are entering a line, you can backspace using CONTROL and the left-arrow key, repair the fault, and then use CONTROL and right-arrow to get back to where you were.It's important to remember that *no change is made permanently until you press RETURN.* If you forget this, and just move the cursor down, it may look repaired on the screen, but when you list you will see the original version.

(5) After pressing RETURN on an edit, the cursor will move to the next line down. If this is a blank space, all is well. If the cursor lands on another line, or on the word READY, however, you will still be editing! Move the cursor clear of all words on the screen before you attempt to type a command, such as LIST. It's always a good idea to LIST a line after editing it, just to make sure that the changes have been carried out.

(6) *Anything* that appears on the screen can be edited. You can type new line numbers. You can take a direct command (with no line number), and add a line number to it. You can also delete error messages. Suppose, for example, you press RETURN, and promptly get an error message because of a fault in a line. Place the cursor at the start of the error message, release the CONTROL key, then press SHIFT and DELETE. This removes the error message. Now move the cursor to the line that contains the error, and mend that.

The Atari editing system is one of the best around, but you need practice in using it. Try it out with a few lines of program that don't matter. Once you have mastered the system, entering programs will be a lot more fun.

Appendix B: Error Codes

When an error is detected while a program is running, the Atari stops the program, and displays an error code. You then need to look up the meaning of the code number, and these are explained only briefly in the manual. This list explains the codes more fully, but concentrates on the more common mistakes. If you make unusual mistakes, it's more obvious!

2 means *not enough memory*. You will get this during program entry if the program is very long. You can also get it when you RUN a program because you have dimensioned a string to a very large size, more than the computer has memory for.
3 means *wrong value*. A number that you are using is outside the range that ought to be used.
4 means *too many variables*. You are limited to 128 variable names.
5 means *string too long*. You have made a string longer than the size you dimensioned.
6 means *not enough DATA*. You have used READ too often, more than you had DATA items for.
8 means *you have an INPUT with a number variable, and you have tried to enter a string.*
9 means *something wrong with dimensioning*. This usually means that you have forgotten to dimension a string or array.
11 means *crazy number*. You get this one if you try to divide a number by zero, or if an operation gives an impossibly large or small number.
12 means *no such line*. You have a GOTO or GOSUB to a line that you forgot to type.
13 means *something wrong with FOR ... NEXT*. You may have the NEXT with no FOR, or you have something like a NEXT N following a FOR J.
13 means *you have a RETURN, but no GOSUB!*
17 means *rubbish in the way*. Part of a line just doesn't make sense. You often get this when you have edited badly, and left something like LIST 2Ø,4Ø at the end of a line.

There are many more error codes, particularly if you are using cassette or disk data files but, by the time you start to get these, you will know enough about your computer to realise what is wrong and how to remedy it. The codes I have dealt with here are the ones that the beginner is most likely to see.

Index

Adapter, 2 to 1, 4
Alphabetical order, 56
Alternate set, 74, 81
Amplitude, 96
Array, 58
ASC, 54
ASCII code, 48
Assignment, 27
Asterisk, 19

Backslash, 19
BASIC, 18
Bass note, 97
Bent-arrow, 27
Border colours, 71
Bouncing-ball, 77
BREAK, 8, 12
Break loop, 38
Brightness, 70
BYE, 10
Bytes, 28

Cartridge slot, 11
Centring words, 25
Channel, 63
Channel number, 97
CHR$, 54
CIRCLE, 93
CLEAR, 14
Clear screen, 27
CLOAD, 15
CLOSE, 63
COLOR, 72, 85
Colour, 65
Colour changes, 8
Colour fill, 91
Colour TV, 1
Coloured characters, 82
Columns, 22
Comma, 22
Comparing strings, 56
Computer, 1
Concatenation, 31
CONTROL, 12
Core program, 107
Counting, 35
CSAVE, 15
Curly bracket, 27
Cursor, 5
Cursor control characters, 75
Cursor control codes, 68

DATA, 45
Data saving, 62
Database, 103
Decrementing, 35
Default colours, 79
Default settings, 71
DEG, 77
DIM, 59
Dimension, 59
Dimensioning, 29
DIN plug, 2
Direct mode, 17
Disks, 64
Display memory, 66
Distortion number, 97, 99
DRAWTO, 83, 87

Editing, 118
Engine noise, 101
Error codes, 120
ESC, 12
Extension lead, 4

File program, 114
Filename, 64
FOR, 39
Forbidden operation, 30
Foundation of program, 106
Frequency, 96
Full screen, 79

GET, 47
GOSUB, 48
GOTO, 38
Graph paper, 88
Graph plotting, 77
Graphics, 65
Graphics mode, 66
Graphics modes, 83
Graphics shapes, 68
Gunshot sounds, 101

Hardware, 2
Hashmark, 32
HELP, 11
Hertz, 96
Hiding messages, 55
High resolution, 65, 83

IF, 42
Incrementing, 35
Inner loop, 40
INPUT, 33
Instruction words, 17
Instructions, 107
Inverse video, 12, 68

Joining, 31

Kaleidoscope, 91
Keyboard, 2, 8

Leader, 14
Line numbers, 14, 18
LIST, 15
LOCATE, 77
Long string, 59
Loop, 38
Loudness, 97
Low resolution, 65

Lower-case, 9
LPRINT, 21
Luminance, 70

Machine code, 65
Mains sockets, 4
Mathematical operations, 35
Memory units, 28
Mode 0, 67
Mode 1, 78
Modulator, 5
Monitor, 70
Mugtrap, 43

Nested loop, 40, 61
Never-ending loop, 38
NEW, 14
NEXT, 39
Not equal sign, 42
Number functions, 36
Number-guessing game, 44
Number totalling, 41
Numbers, 35

ON/OFF switch, 2
Opening channel, 47
OPTION, 10
Outer loop, 40
Outline plan, 104

Padding action, 59
Paper printer, 21
Parallel bus, 16
Peripherals, 1, 13
Phono plug, 3
Pitch, 97
Pitch number, 97
Pixels, 73
Planning, 88
Player-missile graphics, 90
PLOT, 72, 87
Plotting grid, 73
Pointer, 45
POKE, 73
POP, 114
POSITION, 24
Position numbers, 53
Power supply, 2
Precision of numbers, 36
PRINT, 17
Print modifiers, 21
PRINT#6, 80
Program design, 103
Program mode, 17
Program recorder, 12

RAD, 77
Radian, 77
Random lines, 91
READ, 45
READY, 5
Register number, 69
Relative volume, 97
REM, 14, 106
Repeated actions, 38
Reserved words, 17
RESET, 8
Resolution, 65
RESTORE, 47
RETURN, 9, 48
Reverse video, 31
RND, 44

Saving, 62
Screen colour, 69
Self-test, 10
SETCOLOUR, 69
SHIFT, 9
Shopping costs, 46
Sine, 77
Single-key reply, 47
Slicing, 53
Sound, 96
Sound effects, 100
Sound wave, 96
Special keys, 9
Split screen, 79
Sprites, 90
STEP, 39
STR$, 56
String arrays, 59
String function, 52
String variable, 29
Subroutine, 48
Subroutine, design, 108
Subscript number, 58
Subscripted variable, 58
Suppressing cursor, 75

TAB, 23
TAB stops, 23
Tabulation, 23
Terminator, 41
Test, 42
Title, 107
Treble note, 97
Tuning TV, 5
Turtles, 1
TV cable, 3
TV receiver, 3

Underlining, 52
Upper-case, 9

VAL, 56
Variable name, 27, 28

Warbling note, 98
Warning note, 98
Waves breaking, 100
White noise, 100
Working copy, 107

XIO, 92

A Fast, No-nonsense Guide to
The Atari 600XL

Today, everyone needs hands-on experience of computers for success in the modern world. The Atari 600XL is a new version of a well-tried and tested machine, which is excellent for beginners and family use. An immense range of exciting software - from games to business programs - already exists for this micro.

This book makes it easy for you to get the best out of your machine. It shows you how to get rapid results, avoiding all the unnecessary theory, but providing lots of illustrative and varied program examples to make learning a pleasure. You are also shown how to design and write your own programs, and longer program examples are included for you to enjoy and put to practical use.

Get off to a good start with the ATARI 600XL!

Boots, Nottingham, England
Printed in Great Britain 0 246 12370 2

£2.95 net